Copyright

by

Jun Sawada

1999

Formal Verification of an Advanced Pipelined Machine

by

Jun Sawada, B.S., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 1999

Formal Verification of an Advanced Pipelined Machine

Approved by
Dissertation Committee:

_____

_____

_____

_____

_____

_____

To my mother and father,

with gratitude

Acknowledgments

First and foremost, I would like to thank my advisor Warren A. Hunt, Jr. for all the advice during the course of my research. He sparked my interest in the topic studied in this dissertation. His enthusiasm and vision for hardware verification are inspiring.

I am indebted to J Strother Moore and Matt Kaufmann for developing the ACL2 theorem prover and supporting my work throughout the research. I cannot imagine that this work would have ever been possible without their powerful theorem proving system. Bishop Brock's IHS library was invaluable for my research.

I appreciate Bob Boyer for taking time to have numerous discussions on the subject, especially about the correctness of pipelined machines. Harvey Cragon should be credited for teaching me the techniques used in the pipelined machine designs. I also thank Don Fussell for serving as my supervising professor, and Jacob Abraham and Allen Emerson for giving me expert advise on testing and model checking.

I was lucky to have a number of fellow graduate students who were very supportive of my work. Especially, I thank Pete Manolios, Richard Trefler, Rajeev Joshi and Nina Amla for reading my dissertation and helping me to improve the material. I had invaluable discussions with Rob Sumner and Keder Namjoshi on microprocessor designs and verification techniques. Finally, my years in the graduate school was enjoyable largely due to my other friends who were supportive during

my research.

This research was supported in part by the Semiconductor Research Corporation under contract 98-DJ-388.

Jun Sawada

The University of Texas at Austin
December 1999

Formal Verification of an Advanced Pipelined Machine

Publication No. _____

Jun Sawada, Ph.D.

The University of Texas at Austin, 1999

Supervisor: Warren A. Hunt, Jr. and Donald Fussell

The objective in this dissertation is to demonstrate that we can formally verify the correctness of a microprocessor with complex control mechanisms. For the purpose of this research, we designed a new microprocessor model called FM9801, which is a pipelined microprocessor with a number of performance-oriented features: out-of-order issue and completion of instructions using Tomasulo's algorithm, speculative execution with branch prediction, memory optimizations such as load-bypassing and load-forwarding, precise exceptions and interrupts, and context switching between supervisor/user mode. The FM9801 has the capability of executing self-modifying programs as well.

The verification of a pipelined microprocessor is not as simple as the verification of a non-pipelined microprocessor, because the pipelined machine starts the execution of an instruction before completing a previous one. In some cases, a pipelined implementation may execute instructions out of program order or execute them speculatively. The difference in the style of execution between the ideal sequential model and actual implementations makes it difficult to verify or even state the correctness of pipelined microprocessor designs. In this dissertation, we address

what we mean by the "correct" pipelined implementations.

Our verification techniques for the FM9801 is the main topic of this dissertation. One key idea in our approach is the use of the MAETT intermediate abstraction, which is a list of instructions executed by our pipelined microprocessor implementation. Using this abstraction, we were able to directly reason about the executed instructions, which in turn permitted the verification of the entire microprocessor model.

We have verified the FM9801 in two steps. In the first step, we verified an invariant condition defined on the MAETT abstraction. In the second step, we used the verified invariant as an assumption and proved our correctness criterion. We will discuss how this will decompose the verification problem both temporally and spatially. The FM9801 verification is mechanically checked with the ACL2 theorem prover.

Contents

Acknowledgments **v**

Abstract **vii**

Chapter 1 Introduction **1**

Chapter 2 Related Work **4**

Chapter 3 ACL2 Theorem Prover **9**

3.1 Functions and Theorems in ACL2 9

3.2 IHS Library and other ACL2 macros 12

3.3 Data Structures in ACL2 15

3.4 Infix Notation 18

Chapter 4 Verification of a Simple Pipelined Machine **22**

4.1 A Three-Stage Pipelined Machine and Its Correctness 22

4.2 Intermediate Abstraction and Invariant 28

4.3 Proving the Commutative Diagram 34

Chapter 5 Machine Specification of the FM9801 **38**

5.1 Basic Components of the FM9801 39

5.1.1 Address and Data Word 39

5.1.2 Program counter 40
5.1.3 Register Files 40
5.1.4 Memory 42
5.1.5 Efficient Memory Model 43
5.2 Instruction-Set Architecture of the FM9801 45
5.2.1 Instruction-Set Architecture State 45
5.2.2 The Instruction Set of the FM9801 46
5.2.3 Exceptions and Interrupts in the FM9801 52
5.2.4 Formal Definition of the ISA 55
5.3 Microarchitectural Design of FM9801 58
5.3.1 Instruction Fetch Unit and Dispatch Queue. 60
5.3.2 Tomasulo's Algorithm 62
5.3.3 Register Reference Table 62
5.3.4 Reservation Station and Common Data Bus 64
5.3.5 Execution Units 66
5.3.6 Reorder Buffer 66
5.3.7 Speculative Execution 67
5.3.8 Implementation of Exceptions and Interrupts 68
5.3.9 Memory Access by the FM9801 69
5.3.10 Formal Specification of the FM9801 Microarchitecture 71

Chapter 6 Correctness Criteria for Pipelined Machines 75

6.1 Commutative Diagram 75
6.2 Earlier Approaches for Pipelined Machines 78
6.3 Correctness Criterion for Pipeline Machines 84
6.4 Exceptions and Correctness Criterion 89
6.5 Self-Modifying Code in Pipelined Machines 92

Chapter 7 Intermediate Abstraction **96**

7.1 Purpose of an Intermediate Abstraction 96

7.2 Data-Structure and Functions for MAETT 98

7.3 Representation of Instructions 104

7.3.1 Stages of Instructions 104

7.3.2 ISA States and Interrupt Signals 105

7.3.3 Speculatively Executed Instructions 109

7.3.4 Modified Instructions 111

7.3.5 Other INST Fields 112

7.4 Instruction Order 113

7.5 Specifying Instructions by Stages 115

7.6 Last Register Modifiers 118

Chapter 8 Definition and Verification of Invariant Properties **122**

8.1 Definition of the Invariant Condition 122

8.1.1 Overview 122

8.1.2 Weak Invariants 126

8.1.3 Order of Instruction Fetch, Dispatch and Commit 127

8.1.4 Order of Instructions in the Dispatch Queue 129

8.1.5 Order of Instructions in the Reorder Buffer 130

8.1.6 Orders of Load and Store Instructions 132

8.1.7 Absence of Stage Conflicts 133

8.1.8 Absence of Conflicts in the Reorder Buffer 134

8.1.9 Speculatively Executed Instructions 135

8.1.10 Abandoning Speculatively Executed Instructions 136

8.1.11 Stage of Interrupted Instructions 136

8.1.12 Correctness of Intermediate Values 137

8.1.13 Correct Tags in Reservation Stations 139

8.1.14 Tags in Register Reference Table 141
8.1.15 Correct States of Programmer Visible Components 142
8.1.16 Other Invariant Conditions 146
8.2 Verification of the Invariant Condition 147
8.2.1 Overview 147
8.2.2 Verification of Intermediate Values 149
8.2.3 Correctness of Forwarded Data Values 151
8.2.4 Verification of Load-Forwarding and Load-Bypassing 153
8.2.5 Summary 156

Chapter 9 Proof of Correctness Criterion **157**

Chapter 10 Verification Summary **166**
10.1 Cost Analysis 166
10.2 Detected Design Faults 168
10.2.1 Overview 168
10.2.2 Details of Design Faults 169
10.3 Summary 178

Chapter 11 Conclusion **179**

Appendix A **183**
A.1 Proof of Theorem 1 183
A.2 Theorem of Burch and Dill's Diagram Formation 184

Appendix B FM9801 State Definition **187**
B.1 Definition of Words 187
B.2 Definition of Register Files 188
B.3 Definition of the ISA state 189
B.4 Definition of the MA state 190

B.5 Definition of the MAETT state 197

Appendix C List of INST Functions **199**

Appendix D ACL2 Books for the FM9801 Verification **202**

D.1 Basic Books for FM9801 Verification 203

D.1.1 absolute-path.lisp 203

D.1.2 IHS.lisp 204

D.1.3 trivia.lisp 206

D.1.4 define-u-package.lisp 208

D.1.5 utils.lisp 208

D.1.6 b-ops-aux-def.lisp 212

D.1.7 b-ops-aux.lisp 213

D.2 Machine Definitions 225

D.2.1 basic-def.lisp 225

D.2.2 ISA-def.lisp 240

D.2.3 MA2-def.tex 255

D.3 Intermediate Abstraction 314

D.3.1 MAETT-def.lisp 314

D.4 Invariant Definitions 343

D.4.1 invariants-def.lisp 343

D.5 Shared Lemmas 384

D.5.1 MA2-lemmas.tex 384

D.5.2 MAETT-lemmas1.tex 391

D.5.3 MAETT-lemmas2.tex 476

D.5.4 MAETT-lemmas.lisp 607

D.6 Invariant Proofs 608

D.6.1 memory-inv.lisp 608

D.6.2 modifier.lisp 616
D.6.3 wk-inv.lisp 649
D.6.4 in-order.lisp 653
D.6.5 MI-inv.lisp 692
D.6.6 reg-ref.lisp 832
D.6.7 ISA-comp.lisp 903
D.6.8 misc-inv.lisp 959
D.6.9 uniq-inv.lisp 980
D.6.10 invariant-proof.lisp 1082
D.7 Correctness Proof 1085
D.7.1 correctness.lisp 1085

Bibliography **1103**

Vita **1113**

Chapter 1

Introduction

The quality of microprocessors is critically important in today's society, because computers are used in every aspect of our life. The cost of a single bug in a microprocessor design can be significant, since millions of microprocessors are manufactured based on the same design. Testing and simulations are widely used techniques to detect design faults. However, they do not eliminate the possibilities of hidden design flaws in the hardware. At best, simulations and testing can only reduce the number of design faults, but they do not guarantee that the hardware is correctly implemented.

Formal verification is an alternative technique that mathematically proves that a hardware design has no design faults, or in case the hardware design is flawed, it reveals where the design faults exist. Given the soundness of the employed formal verification tools and the accuracy of the verified hardware model, the formal verification can guarantee that no hidden design flaws exist in the design.

Recently, industrial microprocessors are becoming increasingly complex and huge, with many performance optimizing features implemented. Pipelining is a key technique in modern microprocessor designs. It temporally overlaps the execution of instructions in order to improve the throughput. However, it is a cause of the

complexity of microprocessor designs, making verification tasks more difficult.

There have been a number of studies to apply formal verification techniques to pipelined microprocessor designs. However, formally verified microprocessor models are often oversimplified. Today's pipelined microprocessors are very complex machines, which may execute instructions out of program order or sometimes speculatively. None of the research in the past has verified the entire design of a microprocessor with such features.

Our objective in this dissertation is to demonstrate that we can formally verify a microprocessor model with complex control mechanisms. For the purpose of this research, we designed a new microprocessor model called FM9801. The FM9801 is a pipelined microprocessor which implements a number of features: out-of-order issue and completion of instructions using Tomasulo's algorithm, speculative execution with branch prediction, memory optimizations such as load-bypassing and load-forwarding, precise exceptions and interrupts, and context switching between supervisor/user mode.

The definition of correct pipelined microprocessors is one major topic of this dissertation. For microprocessors that execute instructions sequentially, we only have to verify that individual instructions are executed correctly because sequential execution processes instructions one-by-one. This is not the case for pipelined microprocessors, whose implementation may start the execution of an instruction before completing the previous one. Sometimes they may execute instructions out of program order, or may execute them speculatively and later undo the results. The difference in the style of execution between the ideal sequential model and actual implementations makes it difficult to verify or even state the correctness of pipelined microprocessor designs. Thus, we need to establish what we mean by "correct" before proceeding to the verification of the FM9801.

We verify the FM9801 using the ACL2 theorem prover system. ACL2 is both

a programming language and a theorem proving system. Not only we can model and simulate a microprocessor design using ACL2 as a programming system, but we can also prove properties about the microprocessor model using its theorem proving engine. The use of mechanical verification tools such as ACL2 is necessary to avoid human errors and automate the verification process.

The verification techniques for pipelined microprocessors is the main topic of this dissertation. One key idea in our approach is the use of the intermediate abstraction called MAETT. An intermediate abstraction itself is a widely used technique for formal verification. However, our MAETT abstraction is unique in the sense that it builds the history of instructions executed by the pipelined microprocessor. Using this abstraction, we can directly reason about the executed instructions. This eases the verification of machine properties and eventually the correctness of the entire microprocessor model.

The organization of the dissertation is as follows. First we discuss the background of this work in Chapter 2. After discussing the ACL2 logic and notations used in this dissertation in Chapter 3, we present the verification of a simple 3-stage pipelined machine to illustrate our verification techniques in Chapter 4. In Chapter 5, we introduce the FM9801 microprocessor design. In Chapter 6, we discuss the correctness of pipelined machines and introduce our correctness criterion which we use later in the dissertation. In Chapter 7, we construct the intermediate abstraction of the FM9801. This abstraction serves as a foundation for our verification techniques. In Chapter 8, we use this abstraction to define a number of properties which must be satisfied by the FM9801. These properties are verified by the theorem prover one-by-one, assuring that each pipelined machine component is implemented correctly. The verification results of these properties are combined to form the final correctness theorem in Chapter 9. In Chapter 10, we present an overview of the mechanical proof. Finally, we conclude the dissertation in Chapter 11.

Chapter 2

Related Work

Formal verification techniques used in practice can be broadly categorized into algorithmic approaches and theorem proving. The two most commonly used algorithmic techniques are equivalence checking and model checking. Equivalence checking decides whether two combinational circuits implement the same boolean function. Even though the equivalence checking can verify large combinational circuits, it cannot be applied to state holding devices. Model checking [CE81, QS82, CES86] is a procedure to determine whether a state transition system satisfies a property specified as a temporal logic formula. In particular, symbolic model checking [McM93] efficiently represents the set of states by BDDs [Bry86], making it possible to verify systems with large state spaces. However, model checking may suffer an exponential blowup in the number of state variables.

The second approach uses a theorem prover, which is a computer program that can mechanically check some mathematical assertions. An approach based on theorem proving techniques is typically less automated than an algorithmic approach, but it can be applied to large hardware designs with many state holding devices. Because of this reason, verification of microprocessor designs were first attempted using theorem provers [Coh87, Hun94]. We believe theorem provers are

still the only formal verification techniques that can handle sizable microprocessor designs, even though it is possible to combine algorithmic approaches with theorem provers.

Of many publicly available theorem proving systems [GMW79, ORSvH95, GM93, MW97, CAB+86], we use the ACL2 theorem provers [KM96]. What makes ACL2 distinct from other theorem provers is that it is not only a theorem proving system but also a programming environment. This allows us to both simulate and prove properties about microprocessor models defined in ACL2. There have been a number of hardware verification projects carried out using ACL2[BKM96, BH97, Rus97, WGH98, MLK98, Rus98].

The verification of microprocessors was pioneered in the FM8501 [Hun94] and the Viper project [Coh87]. These studies were reproduced and extended in the following research projects. One mile stone was the FM9001 project [HB92]. This microprocessor design is specified at 4-levels. The highest level is the instruction-set specification while its lowest layer is the net-list of the actual hardware implementation. By proving that each layer is a correct implementation of the layer immediately above, they verified the actual microprocessor correctly implements the instruction-set specification. The microprocessor design of the FM9001 is not pipelined.

The verification of pipelined machines has been also studied in a number of projects. One of the earliest studies was carried out by Srivas and Bickford who verified the Mini-Cayuga [SB90]. Bronstein and Talcott [BT90] also verified a pipelined machine using Nqthm theorem prover. In these studies, the mapping functions, which became known as *skewed abstraction function*, are used to map multiple pipelined states at different moments to a single sequential state. The skewed abstraction function for pipelined machines has been used in a number of verification studies [Cyr93, TK94, Coe94, WC95, AL95]. The idea has also been applied to the verification of a commercial microprocessor in the AAMP5 verification

project [SM95].

The problem of the skewed abstraction function is its complexity. All timing delays in the pipelined machine should be considered in the definition of skewed abstraction function. Since the correctness theorem is defined using the skewed abstraction function, it complicates the correctness statement itself to the point where it is difficult to assess its validity.

The pipeline flushing diagram introduced by Burch and Dill [BD94] was a solution to this problem. Unlike manually defined skewed abstraction functions, they used the pipelined implementation itself as an abstraction function. In their scheme, they first flush the pipeline by running the microprocessor model without fetching new instructions, and then compare the resulting flushed state with the sequential execution model. The pipeline flushing diagram has been applied to the verification of a Motorola CAP DSP [BH97].

Pipeline flushing diagram can be applied to pipelined microprocessors that execute instructions out of program order and to some superscalar designs [Bur96, WB96]. However, their correctness criterion is not directly applicable to designs with speculative execution nor to those with external interrupts. Since both features are implemented in the FM9801, we need a new correctness criterion to handle these cases. This correctness criterion is discussed in detail in Chapter 6.

Another concept used in Burch and Dill's verification method is *uninterpreted functions*. They syntactically compare the results of pipelined machines represented as expressions including uninterpreted function symbols. Although this technique has been used for a while in the community of theorem provers [Sho84], a number of following studies have applied uninterpreted functions to pipelined machine verifications. Some attempted to improve its verification efficiency using caching [JDB95], while others attempted to encode pipeline execution results with binary decision diagrams [BBCZ98]. Miroslav and Bryant [VB98, VB99] improved the verification

efficiency by dividing terms into P-terms and G-terms, which are encoded using binary decision diagrams. We consider that these studies focusing on improving verification engines are orthogonal to our work. They attempt to verify an entire microprocessor model without decomposing the verification problem. Rather, our research focus is decomposing the verification problems into a number of subproblems which can be handled by existing verification tools.

The incremental flushing technique introduced by Skakkebæk et al. decomposes the commutative diagram into small steps that flush one instruction at a time [SJD98]. Hosabettu et al. [HSG98, HGS99] decomposed microprocessor verification by using so called "completion functions", which calculate the effects of completing partially executed instructions. These approaches break down a commutative diagram involving multiple state transitions into small diagrams involving single state transitions, thereby temporally reducing the complexity of the verification problem. However, they do not spatially decompose the problem because they directly analyze the state transition of the entire microprocessor.

Compositional model checking decomposes the verification problem with respect to components, spatially reducing the size of the verification problem. McMillan [McM98] used compositional model checking to verify out-of-order execution core with Tomasulo's algorithm. A model checker is used to independently verify several conditions about inter-component signals. This effectively breaks down the verification of the entire machine design into the verification of components. The verified machine model is an execution core of a microprocessor and it does not implement speculative executions nor exceptions. Similar work is reported by Henzinger et al. using the assume-guarantee method. [HQR98]

Tomasulo's algorithm verification by Damm and Pnueli [DP97] uses a generalized machine that executes instructions nondeterministically an an intermediate abstraction model. Their technique is similar to the intermediate abstraction dis-

cussed in this dissertation, because they use a list of instructions in the program to define the semantics of the generalized machine. However, it is unknown whether a similar generalization can be defined for a more complex pipelined machine which implements branching and speculative executions.

Chapter 3

ACL2 Theorem Prover

3.1 Functions and Theorems in ACL2

ACL2 is a theorem prover system as well as a programming environment. Users can define and execute functions, using the ACL2 logic as a programming language. Users can also prove theorems using the ACL2 theorem prover. In this section, we summarize how to define functions and prove theorems in the ACL2 system.

The ACL2 logic implements a subset of Common Lisp[GLS90]. The ACL2 logic expresses function applications with a prefix notation, just like Common Lisp. For instance, multiplication of `x` and `y` is represented as `(* x y)` instead of `x * y`.

Functions are defined with `defun` expressions in the same way as in Common Lisp. The ACL2 function is a logical object which we can reason about, at the same time it can be evaluated with concrete arguments. Here is an example ACL2 function definition.

```
(defun factorial (x)
  (if (zp x) 1 (* x (factorial (- x 1)))))
```

This `defun` expression defines `factorial` as a function that takes one argument and returns its factorial number. For example, evaluating `(fact 3)` returns `6`. In

this definition, `factorial` returns `1` if `(zp x)` is true, i.e., argument `x` is not a positive integer. Otherwise, the function first calculates the factorial of `x` minus `1` by calling itself recursively, and then multiplies its result with `x`. Function `(zp x)` is a pre-defined function in the ACL2 logic which is equivalent to

```
(if (integerp x) (<= x 0) T)).
```

Table 3.1 shows some of the basic functions pre-defined in the ACL2 logic.

Some definitions of ACL2 functions use *guards*. A guard restricts the type of legitimate arguments for execution. For instance, the factorial function can be defined as:

```
(defun g-factorial (x)
  (declare (xargs :guards (and (integerp x) (>= x 0))))
  (if (zp x)
      1
      (* x (g-factorial (- x 1)))))
```

The newly defined function `g-factorial` only accepts non-negative integers as arguments for execution. The compiler attached to ACL2 may take advantage of the fact to improve the execution speed of the function. The machine designs described in this dissertation are defined using guards, so that the simulation of the machines runs fast.

Lemmas and theorems in the ACL2 logic are defined with `defthm`. For instance, the following theorem states the associativity of addition.

```
(defthm assoc-+  (equal (+ (+ x y) z) (+ x (+ y z))))
```

When a `defthm` expression is submitted, the ACL2 theorem prover attempts to prove the theorem. When it successfully proves the theorem, ACL2 stores it in the database for the proven theorems. In the ACL2 logic, there is no distinction between lemmas and theorems.

ACL2 Function and Constants	Informal Description	
`T`	True value.	*
`nil`	False value as well as the empty list.	*
`(+ x y)`	$x + y$	*
`(- x y)`	$x - y$	*
`(* x y)`	$x \times y$	*
`(/ x y)`	x/y	*
`(mod x y)`	x **mod** y	*
`(expt x y)`	x^y	*
`(1+ x)`	$x + 1$	*
`(1- x)`	$x - 1$	*
`(< x y)`	$x < y$	*
`(<= x y)`	$x \leq y$	*
`(equal x y)`	x equals y.	*
`(if x y z)`	If x is true, returns y. Otherwise z.	*
`(not x)`	$\neg x$	*
`(and x y)`	$x \wedge y$	*
`(or x y)`	$x \vee y$	*
`(implies x y)`	$x \rightarrow y$	*
`(iff x y)`	$x \leftrightarrow y$	*
`(car x)`	First element of cons pair x.	*
`(cdr x)`	Second element of cons pair x.	*
`(cadr x)`	`(car (cdr x))`	*
`(cddr x)`	`(cdr (cdr x))`	*
`(cons x y)`	Cons pair of x and y.	*
`(null x)`	x is *nil*, i.e., the empty list.	*
`(consp x)`	x is a cons. Note `(consp nil)` is false.	*
`(endp x)`	x is *nil* or an atomic object	*
`(len x)`	Length of list x.	
`(append x y)`	Concatenation of list x and y.	*
`(nth n x)`	The n'th element of list x.	*
`(nthcdr n x)`	Removes the first n elements from list x.	*
`(list x y ...)`	List whose elements are x, y, ...	*
`(integerp x)`	True if x is an integer.	*
`(true-listp x)`	True if x is a list terminating with *nil*.	
`(zp x)`	x is not a positive integer.	

Table 3.1: Description of Basic ACL2 functions and constants. Those with asterisk marks have corresponding definitions in Common Lisp.

The ACL2 prover exploits mathematical induction and term rewriting with heuristics to prove many theorems automatically. However, it is almost always the case that complex theorems cannot be verified automatically. The user has to describe an outline of the proof, by providing intermediate theorems that fill the gap between the axioms and the final theorems. A file containing these intermediate and the final theorems is called a *book*.

3.2 IHS Library and other ACL2 macros

The *Integer Hardware Specification* (IHS) library was written by Bishop Brock originally for Motorola CAP DSP verification project[BH97]. In the IHS library, bit vectors are represented with integers instead of conventional lists of boolean values. As a result, the executions of hardware specifications written in the IHS library are faster than those using list representations of bit vectors. The IHS library also defines numerous theorems about basic bit vector operations, which help the mechanized proofs of theorems about hardware specifications.

In the IHS library, integers represent bits and bit vectors. Integer 1 and 0 represent a bit. A n-bit bit vector is represented by an integer whose binary representation has the same least significant n bits. For instance, a four-bit bit vector 1101 can be represented by integer 13. The IHS library does not provide a method to specify the length of the bit vector represented by an integer. Thus 13 can represent the four-bit bit vector 1101 as well as the 16-bit vector 0000000000001101.

The IHS library defines functions to manipulate bit vectors. Table 3.2 lists the IHS functions which are used in this dissertation. In this table, bit arguments are represented with `a` and `b`, while bit vectors are represented with `u` and `v`.

Simple logical bit operations are defined with `b-not`, `b-and`, `b-ior`, and so

on. For example,

```
(b-not 0) = 1
(b-not 1) = 0
(b-and 1 0) = 0
(b-and 1 1) = 1.
```

Bit-wise logical operators are defined separately. For example, `lognot` takes an integer representing a bit vector and returns the integer representing its 1's complement. Function `logand` returns the bit-wise logical AND of two arguments. For instance,

```
(lognot 0) = -1
(lognot 1) = -2
(logand 3 5) = 1
(logand 3 -1) = 3.
```

The IHS library also defines functions to decompose and combine bit vectors. The most basic functions are `logcar`, `logcdr`, and `logcons`, which are analogous to the Lisp functions `car`, `cdr`, and `cons`, respectively. In the IHS library, a bit vector is viewed as a list of bits, whose first element is the least significant bit. Just like `(car lst)` returns the first element of list `lst` and `(cdr lst)` returns the rest in Lisp, `(logcar v)` returns the least significant bit of `v` and `(logcdr v)` returns the bit-vector without the least significant bit. Function `(logcons b v)` adjoins `b` to `v`, with `b` as the least significant bit of the resulting vector. We show example evaluations of these functions, with Common Lisp representing of binary numbers with the prefix `#b`.

```
(logcar #b1101) = 1
(logcdr #b1101) = #b110
(logcons 1 #b110) = #b1101
```

IHS Functions	Informal Description using the C language
`(bitp b)`	T if `b` is a bit.
`(bfix x)`	Coerce `x` to a bit.
`(zbp b)`	Bit-boolean converter. Nil if b is 1. Otherwise, T.
`(b1p b)`	Bit-boolean converter. T if b is 1. Otherwise, nil.
`(b-if b x y)`	If `b` is 1, return `x`. Otherwise, `y`.
`(b-not a)`	Bit negation.
`(b-and a b)`	Bit AND.
`(b-ior a b)`	Bit inclusive OR.
`(b-xor a b)`	Bit exclusive OR.
`(b-eqv a b)`	Bit equivalence.
`(b-nand a b)`	Bit NAND. `(b-not (b-and a b))`
`(b-nor a b)`	Bit NOR. `(b-not (b-ior a b))`
`(b-andc1 a b)`	`(b-and (b-not a) b)`
`(b-andc2 a b)`	`(b-and a (b-not b))`
`(b-orc1 a b)`	`(b-ior (b-not a) b)`
`(b-orc2 a b)`	`(b-ior a (b-not b))`
`(unsigned-byte-p n v)`	$0 \leq$ `v` $< 2^{\mathtt{n}}$
`(logbit n v)`	The `n`'th bit of bit-vector `v`. `(v >> n) & 0x1`
`(logand u v)`	Bitwise AND. `(u & v)`
`(logcar v)`	The least significant bit of `v`. `(v & 0x1)`
`(logcdr v)`	Bit vector `v` without the least significant bit. `(v >> 1)`
`(logcons b v)`	Concatenation of bit `b` to vector `v`. `(v << 1) | b`
`(logior u v)`	Bitwise inclusive OR. `(u | v)`
`(lognot v)`	1's complement. `(~v)`
`(logxor u v)`	Bitwise exclusive OR. `(u ^ v)`.
`(loghead n v)`	The least significant `n` bits in bit vector `v`. `(v &` $2^{\mathtt{n}} - 1$ `)`
`(logtail n v)`	Bit vector `v` without the least significant `n` bits. `(v >> n)`
`(logextu n m v)`	Sign-extend `m`-bit vector `v` to `n` bits.
`(logapp n u v)`	Concatenation of bit vectors. `u | (v << n)`
`(rdb (cons n i) v)`	`n` bits of `v` from the `i`'th bit. `(v >> i) &` $2^{\mathtt{n}} - 1$

Table 3.2: List of Basic IHS functions. We use C expressions to give informal descriptions of functions.

The IHS library proves various theorems about the bit and bit-vector functions. For example, the IHS library provides the following theorem.

```
(defthm logcar-logcdr-elim
    (implies (integerp i)
             (equal (logcons (logcar i) (logcdr i))
                    i)))
```

Suppose `i` is an integer representing a bit-vector. Taking the least significant bit and the remaining bits of the represented bit vector, and adjoining them will return `i` itself. The IHS library uses this theorem as an ACL2 rewriting rule. When this rewriting rule is activated, ACL2 rewrites a term of the form `(logcons (logcar` x`) (logcdr` x`))` into x, where x can be any ACL2 term representing an integer.

The IHS library defines an extensive set of bit vector functions and theorems. The functions are carefully defined so that the hardware specification using the IHS library can be executed fast. In fact, Brock modeled Motorola's CAP digital signal processor using the IHS library, and this model outperformed a Cadence-based RTL specification[BH97] in simulations. We use the IHS library to specify our machine models, because the IHS library is a good basis for writing a formal hardware specification, on which we perform both simulations and formal verification.

3.3 Data Structures in ACL2

We use many ACL2 structured types in the specification and verification of machine models discussed in this dissertation. These structured types are defined using ACL2 macros supplied in the ACL2 public books. In this section, we discuss such ACL2 macros defining data-structures, namely, `defstructure`, `deflist`, and `defword`.

An ACL2 macro `defstructure` defines a structure type. The closest counterpart in Common Lisp is `defstruct`. ACL2's `defstructure` not only defines the structure type in the same way as Common Lisp's `defstruct`, but it also auto-

matically generates and proves ACL2 theorems associated with the newly defined data-structure.

For example, we will model a cache line that contains a valid bit, an address tag, and data. The new structure type `c-line` can be defined with `defstructure` as follows:

```
(defstructure c-line
  (valid   (:assert (bitp valid)    :rewrite))
  (addr    (:assert (integerp addr) :rewrite))
  (data    (:assert (integerp data) :rewrite)))
```

Structure `c-line` contains fields `valid`, `addr`, and `data`. The keyword `:assert` is followed by a type assertion. Field `valid` holds a bit, while fields `addr` and `data` hold integers representing bit vectors. The keyword `:rewrite` directs `defstructure` to automatically generate type-related rewriting rules, that will be explained shortly.

The `defstructure` shown above defines one constructor function, three accessor functions, and one type predicate. The constructor function for the structure type `c-line` is named `c-line` itself. Structure `c-line` consisting of valid bit `vld`, address `ad`, and data `dt` is defined as `(c-line vld ad dt)`.

Accessor functions `c-line-valid`, `c-line-addr`, and `c-line-data` take a `c-line` structure and return the value in the corresponding field. Type predicate `(c-line-p x)` is true if `x` is a `c-line` structure.

The `defstructure` of `c-line` automatically generates and proves theorems about the newly defined record type. Some of the lemmas are shown in Figure 3.1.

The ACL2 macro `deflist` defines the true-list type. `Deflist` has a syntax of the form `(deflist` $<$ *list-type-name* $>$ $<$ *type* $>$ `)`, which defines a **nil**-terminating list of elements whose type is $<$ *type* $>$. For example,

```
(deflist cache-p c-line)
```

```
;  This lemma simplifies reads of an explicit constructor.
   (DEFTHM DEFS-READ-C-LINE
           (AND (EQUAL (C-LINE-VALID (C-LINE VALID ADDR DATA))
                       VALID)
                (EQUAL (C-LINE-ADDR (C-LINE VALID ADDR DATA))
                       ADDR)
                (EQUAL (C-LINE-DATA (C-LINE VALID ADDR DATA))
                       DATA)))

;  This is the :ELIM lemma for the constructor.
   (DEFTHM DEFS-ELIMINATE-C-LINE
           (IMPLIES (WEAK-C-LINE-P C-LINE)
                    (EQUAL (C-LINE (C-LINE-VALID C-LINE)
                                   (C-LINE-ADDR C-LINE)
                                   (C-LINE-DATA C-LINE))
                           C-LINE))
           :RULE-CLASSES (:REWRITE :ELIM))

;  This lemma captures all assertions about the structure.  This lemma is not
;  guaranteed to prove.  If it does not prove than you may have to provide
;  some :HINTS.  Any :ASSERTION-LEMMA-HINTS option to DEFSTRUCTURE will be
;  attached to this lemma.  Be sure that you have not specified
;  unsatisfiable assertions.
   (DEFTHM DEFS-C-LINE-ASSERTIONS
           (IMPLIES (C-LINE-P C-LINE)
                    (AND (WEAK-C-LINE-P C-LINE)
                         (BITP (C-LINE-VALID C-LINE))
                         (INTEGERP (C-LINE-ADDR C-LINE))
                         (INTEGERP (C-LINE-DATA C-LINE))
                         T))
           :RULE-CLASSES
           ((:REWRITE :COROLLARY
                      (IMPLIES (C-LINE-P C-LINE)
                               (BITP (C-LINE-VALID C-LINE))))
            (:REWRITE :COROLLARY
                      (IMPLIES (C-LINE-P C-LINE)
                               (INTEGERP (C-LINE-ADDR C-LINE))))
            (:REWRITE :COROLLARY
                      (IMPLIES (C-LINE-P C-LINE)
                               (INTEGERP (C-LINE-DATA C-LINE))))))
```

Figure 3.1: Some theorems about the structure type `c-line`. The `defstructure` macro automatically generates these theorems including the comments. `WEAK-C-LINE-P` is a well-formedness predicate for the structure `c-line`.

defines the true-list type whose elements are `c-line` structures. The type predicate is `cache-p` itself.

Macro `defword` defines a word type consisting of multiple fields. The `defword` syntax is of the form:

`(defword` $<$ *word-name* $>$

$(<$ *field*$_1>$ $<$ *width*$_1>$ $<$ *pos*$_1>$`)`

$\vdots$

$(<$ *field*$_i>$ $<$ *width*$_i>$ $<$ *pos*$_i>$`)`

$\vdots$

$(<$ *field*$_n>$ $<$ *width*$_n>$ $<$ *pos*$_n>$`))`

This `defword` defines a word type whose name is $<$ *word-name* $>$. The $<$ *width*$_i>$ bits from the position $<$ *pos*$_i>$ is referred to as the field $<$ *field*$_i>$.

For example, the following `defword` defines the `addr` word.

```
(defword addr
    (page     8 8)
    (offset   8 0))
```

The 8 bits starting from the 8th bit are designated as the `page` field, and the 8 bits starting from the 0'th bit are designated as the `offset` field. The accessor functions to these field are `addr-page` and `addr-offset`. Thus `(addr-page #x1234) = #x12` and `(addr-page #x1234) = #x34`.

3.4 Infix Notation

All the functional definitions and theorems discussed in this dissertation are formally defined and proved by the ACL2 theorem prover. Naturally, they are written in the ACL2 syntax, which is basically the same as the Common Lisp syntax. It is unfortunate that some people find that prefix notation of the Lisp syntax is not

intuitive and moreover hard to read. Therefore, we will use an infix notation of ACL2 formulae in the body of the dissertation. This infix formulae are mechanically generated from the corresponding ACL2 formulae.

In the infix notation, variables are printed in italics. Function application `(f x y z)` is printed with usual notation $\mathrm{f}(x, y, z)$, unless the function is shown in Table 3.4 or the function is an accessor function of a structure type defined by `defstructure`, which will be discussed shortly. Function symbols are printed in Roman. Constants are printed in a typewriter font. Quotation is used in the same way as in the original ACL2 logic. For example, a list is printed like '(a b c). Binary number `#b010` is printed as $\mathtt{010}_2$ and hexadecimal number #xa08 is printed as $\mathtt{a08}_{16}$.

Control structures are printed as follows. An `if`-expression `(if x y z)` is printed as

if *x* **then** *y* **else** *z* **fi** .

A `cond`-expression `(cond ((test1 exp1) (test2 exp2) (t exp3)))` is printed as:

if *test1* **then** *exp1* **elseif** *test2* **then** *exp2* **else** *exp3* **fi** .

A `let`-expression `(let ((v1 exp1) (v2 exp2)) body)` is printed as:

let *v1* **be** *exp1*, *v2* **be** *exp2* **in** *body* .

And `(let* ((v1 exp1) (v2 exp2)) body)` is printed as:

let* *v1* **be** *exp1*, *v2* **be** *exp2* **in** *body* .

Both `let` and `let*` forms are used to bind local variables. Bindings occur in parallel in a `let` form, while bindings occur sequentially in a `let*` form.

For example, the definition of the function `factorial` and the theorem `assoc-+` given in Section 3.1 are printed as follows:

DEFINITION:
factorial (x)
$\stackrel{def}{=}$
if $x \simeq \mathtt{0}$ **then** `1`
else $x \times$ factorial $(x - \mathtt{1})$
fi

THEOREM: assoc-+
$((x + y) + z) = (x + (y + z))$

Structure definition `defstructure` and true list definition `deflist` are also printed out specially. For example, the definition of structure type `c-line` and true-list type `cache` in the previous section are printed as:

Defstructure c-line {
bitp valid ;
integerp addr ;
integerp data ;
}

Deflist cache-p **as List of** c-line

Accessor functions to the structure fields are printed as suffix operators in the infix notation. For example, the value in the field `valid` is defined in `(c-line-valid x)` in the ACL2 syntax. In the infix notation, it is printed as x.valid.

ACL2 Syntax	Infix Syntax
`t`	**t**
`nil`	**nil**
`(not x)`	$\neg\, y$
`(or x y)`	$x \vee y$
`(and x y)`	$x \wedge y$
`(iff x y)`	$x \leftrightarrow y$
`(implies x y)`	$x \rightarrow y$
`(+ x y)`	$x + y$
`(- x y)`	$x - y$
`(* x y)`	$x \times y$
`(mod x y)`	x **mod** y
`(/ x y)`	$x \,/\, y$
`(1+ x)`	$x + 1$
`(1- x)`	$x - 1$
`(integerp x)`	$x \in$ **Int**
`(append x y)`	x @ y
`(member-equal x y)`	$x \in y$
`(not (member-equal x y))`	$x \notin y$
`(>= x y)`	$x \geq y$
`(> x y)`	$x > y$
`(<= x y)`	$x \leq y$
`(< x y)`	$x < y$
`(equal x y)`	$x = y$
`(not (>= x y))`	$x \ngeq y$
`(not (> x y))`	$x \not> y$
`(not (<= x y))`	$x \not\leq y$
`(not (< x y))`	$x \not< y$
`(not (equal x y))`	$x \neq y$
`(zbp x)`	$x =$ `0`
`(b1p x)`	$x \neq$ `0`
`(not (zbp x))`	$x \neq$ `0`
`(not (b1p x))`	$x =$ `0`
`(INST-in x y)`	$x \in_{\mathrm{MT}} y$
`(INST-in-order-p x y z)`	x **precedes** y **in** z
`(tag-in-order x y z)`	$x <_{\mathrm{tag}} y$ **in** z

Table 3.3: The list of infix functions. The last seven functions are not built-in ACL2 functions. They are defined during the verification of the FM9801.

Chapter 4

Verification of a Simple Pipelined Machine

In this chapter, we present the verification of a simple three-stage pipelined machine. This will serve as an overview of our pipeline verification techniques that we later use to verify more complex pipelined machine named FM9801.

4.1 A Three-Stage Pipelined Machine and Its Correctness

Pipelining is a key idea in the design of modern microprocessors. It improves the performance of microprocessors by overlapping the execution of instructions. A pipelined microprocessor typically starts the execution of an instruction before the completion of the previous instruction. However, programmers imagine that microprocessors execute instructions one-by-one. This makes it natural to define the specification of a microprocessor as a sequential execution model. Therefore, the verification of a pipelined microprocessor needs to prove that the pipelined implementation appears to behave as its sequential execution model does.

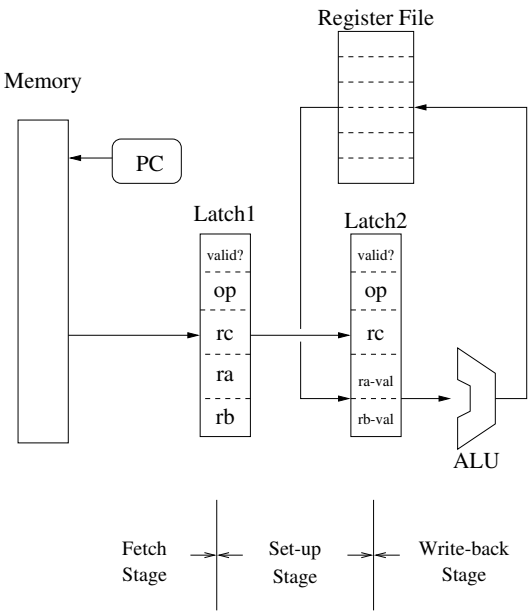

Figure 4.1: The three-stage pipelined machine.

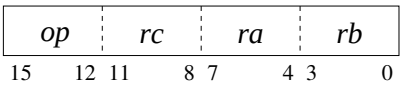

Figure 4.2: The instruction format for the three-stage pipelined machine.

To study this problem, we will consider a three-stage pipelined machine. Figure 4.1 shows its block diagram. This machine consists of a program counter (PC), a register file, memory, an ALU, and two pipeline latches. The register file is a collection of registers.

The instruction format for this machine is shown in Fig. 4.2. An instruction is a 16-bit word and it has four fields: opcode field *op*, destination register field *rc*, source register fields *ra* and *rb*. The bits between the 12th bit and the 15th bit are the opcode, which specifies the instruction type. The opcodes of ADD and SUB instructions are 0 and 1, respectively. An instruction with opcode other than 0 and 1 is considered to be a NOP, which only increments the program counter.

There are three stages in the pipeline: the fetch stage, the set-up stage,

and the write-back stage. The machine fetches an instruction in the fetch stage, reads source registers in the set-up stage, and performs an arithmetic operation and updates the destination register in the write-back stage. The latches are used to store intermediate results. The *valid?* flag of each latch is set to 1 when an instruction occupies the latch. The *op* field stores the opcode of the stored instruction, and the *rc*, *ra*, and *rb* fields store the operand register designators. The *ra-val* and *rb-val* fields store the two source operand register values.

Let us consider the execution of the following three instructions. In this program, the operand registers *rc*, *ra* and *rb* are printed in that order. For example, i_0 is an ADD instruction that reads registers R1 and R3 and stores the results in R2.

i_0: ADD R2, R1, R3

i_1: SUB R4, R2, R5

i_2: ADD R7, R5, R6

Table 4.1 shows the latches in which the intermediate results of instructions are stored at each time. For example, the instruction i_0 is fetched between time 0 and 1, it goes through the set-up stage between time 1 and 2, and it finishes the write-back stage between time 2 and 3. Thus, the intermediate results of i_0 are stored in latch1 at time 1 and in latch2 at time 2.

Table 4.1: A reservation table for the three-stage pipelined machine.

Time	0	1	2	3	4	5	6
i_0		latch1	latch2				
i_1			latch1	latch1	latch2		
i_2					latch1	latch2	

This table shows a typical pipelined execution. While i_0 is at the set-up stage between time 1 and 2, instruction i_1 is fetched simultaneously. Since the instruction i_1 uses the value of register R2 which is the result of instruction i_0, instruction i_1

must wait for i_0 to update R2 before reading its value in the set-up stage. Thus, the instruction i_1 *stalls* between times 2 and 3. After i_0 completes the write-back stage at time 3, i_1 continues the rest of the execution in the set-up and write-back stages. Instruction i_2 is executed between times 3 and 6.

Because the execution of instructions is overlapped, a pipelined machine state may not correspond to any state which programmers expect to see. For example at time 3, the program counter points to the next instruction to be fetched, namely i_2. However, the register file records the result of i_0, but not i_1 yet since i_1 is still at latch1. In other words, the program counter appears as if we have completed two instructions i_0 and i_1, but the registers appear as if we have only completed the instruction i_0. Thus, the pipeline state at time 3 does not correspond to any state observable by executing instructions sequentially.

To be more concrete, we define the machine in the ACL2 logic at two levels: the *instruction-set architecture* (ISA) level and the *microarchitecture* (MA) level. The ISA models the machine behavior that programmers have in mind. It executes instructions one at a time. This style of execution is called *sequential execution*. On the other hand, the MA model defines how the actual pipelined machine behaves. Instruction executions are overlapped in this model, and this style of execution is called *pipelined execution*.

The behavior of the ISA model is given by the following function.

```
DEFINITION:
ISA-step (ISA)
def
=
let inst be read-mem (ISA.pc, ISA.mem)
     in
let op be op-field (inst),
    rc be rc-field (inst),
    ra be ra-field (inst),
    rb be rb-field (inst)
     in
if op = 0 then ISA-add (rc, ra, rb, ISA)
 elseif op = 1 then ISA-sub (rc, ra, rb, ISA)
```

else ISA-default (ISA)
fi

This function takes the current state ISA and returns the new state after executing one instruction. The program counter and the memory in state ISA are represented as ISA.pc and ISA.mem using the suffix operator discussed in the last chapter. ISA-step reads an instruction *inst* from the memory ISA.mem at the location addressed by the program counter ISA.pc, divides it into instruction fields, and executes it appropriately depending on the opcode. For example, if the op-field of *inst* contains 0, the ISA-step function performs an ADD instruction whose effect is defined by the function ISA-add. Similarly, ISA-sub and ISA-default define the effects of the SUB and NOP instructions, respectively.

We can define a recursive function ISA-stepn(ISA, n), which calculates the result of executing n instructions. It is defined to apply ISA-step repeatedly n times.

DEFINITION:
ISA-stepn (ISA, n)
$\stackrel{def}{=}$
if $n \simeq 0$ **then** ISA
else ISA-stepn (ISA-step (ISA), $n - 1$)
fi

The function MA-step(MA, sig) takes the current pipeline state MA and an external input signal *sig*, and returns the pipeline state at the next clock cycle. It defines the behavior of the pipelined machine at the MA level, by specifying how individual components behave in every clock cycle.

DEFINITION:
MA-step (MA, *sig*)
$\stackrel{def}{=}$
MA-state (step-pc (MA, *sig*),
step-regs (MA),
MA.mem,
step-latch1 (MA, *sig*),
step-latch2 (MA))

Functions step-pc, step-regs, step-latch1, and step-latch2 define the new state of the program counter, the register file, and pipeline latches latch1 and latch2. The memory state does not change. Constructor function MA-state combines these component states to form the new MA state. The recursive function MA-stepn(MA, *sig-list*, n) applies MA-step repeatedly n times and returns the MA state after n clock cycles later. The argument *sig-list* is a list of input signals.

DEFINITION:
MA-stepn (MA, *sig-list*, n)
$\stackrel{def}{=}$
if $n \simeq 0$ **then** MA
else MA-stepn (MA-step (MA, car (*sig-list*)), cdr (*sig-list*), $n - 1$)
fi

One key idea in comparing pipelined machine states to sequential execution states is using *pipeline flushed states*. An MA state is a pipeline flushed state if no instructions are partially executed in the pipeline. In Table 4.1, the MA is in pipeline flushed states at time 0 and 6. In a pipeline flushed state, all programmer visible components, such as the program counter, the register file, and the memory, are synchronized. Thus it is easy to define the corresponding ISA state for a pipeline flushed state. We define this correspondence as a projection function proj(MA), which returns an ISA state by extracting the program counter, the register file, and the memory states from the pipeline state MA.

DEFINITION:
proj (MA) $\stackrel{def}{=}$ ISA-state (MA.pc, MA.regs, MA.mem)

Figure 4.3 shows the commutative diagram that represents the correctness of our pipelined machine. Consider an initial state MA_0, which we assume is a pipeline flushed state. There are two paths to follow in the commutative diagram. One path runs the MA model for n steps where n is an arbitrary natural number. Suppose the final state MA_n is also a flushed state. Then we can map the final state MA_n to

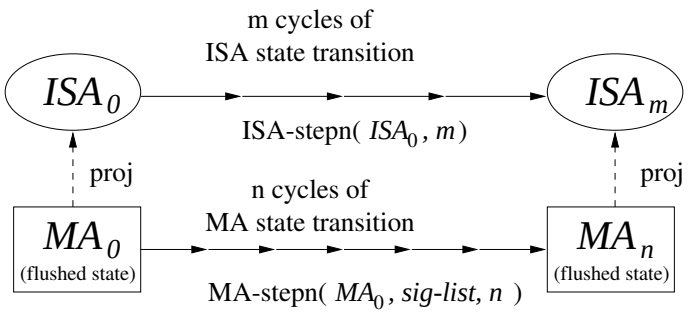

Figure 4.3: The commutative diagram for the pipelined machine.

ISA_m with the projection function. Let m be the number of instructions executed during the state transition from MA_0 to MA_n. The other path first projects the initial state MA_0 to ISA_0 and runs the ISA for m cycles to reach ISA_m. If the MA correctly implements the ISA, the same ISA_m must be obtained by following the two paths. We will prove this commutative diagram in the rest of the chapter.

4.2 Intermediate Abstraction and Invariant

It is often difficult to directly verify an entire pipelined machine. Our example machine is simple, but a typical pipelined microprocessor has a long pipeline with a complex control logic. Instead of directly analyzing the entire microarchitecture, we show that each instruction is executed correctly. This allows us to verify the machine design incrementally, and later combine the results together to prove the commutative diagram.

To pursue this idea, our verification approach first defines an intermediate abstraction, which builds a list of completely executed instructions and in-flight instructions. For example at time 4 in Table 4.1, instruction i_0 has been completely executed, and i_1 and i_2 are in-flight. The intermediate abstraction represents the MA state at time 4 with a list of instructions i_0, i_1 and i_2.

More precisely speaking, the status of each instruction is recorded in the

intermediate abstraction. We represent the status of an instruction with a structure type named INST. In ACL2, the structure can be defined with the `defstructure` macro discussed in the last chapter.

```
Defstructure INST {
    stage-p          stg ;          // Current Stage
    ISA-state-p      pre-ISA ;      // Pre-ISA state
    ISA-state-p      post-ISA ;     // Post-ISA state
    }
```

This structure has three fields *stg*, *pre-ISA*, and *post-ISA*. Field *stg* represents the current stage of the represented instruction, and *pre-ISA* and *post-ISA* store ISA states which we will describe shortly. Fields values of INST structure i are represented as i.stg, i.pre-ISA and i.post-ISA.

Let i_k^t denote the INST structure representing the status of instruction i_k at time t in Table 4.1. Since i_0 is at latch1 at time 1, i_0^1.stg = `'latch1`. Similarly, i_0^2.stg = `'latch2`. The stage of completed instructions is defined as `'retire`, so i_0^3.stg = `'retire`.

Using this instruction representation, we define the intermediate abstraction state. We call this intermediate abstraction *Microarchitecture Execution Trace Table* (MAETT)[SH97]. It is defined using the ACL2 structure:

```
Defstructure MAETT {
    ISA-state-p      init-ISA ;     // Initial ISA state
    INST-listp       trace ;        // List of Executed Instructions
    }
```

The *trace* field stores the list of completed and in-flight instructions. Let MT_t be the MAETT for the MA state at time t in Table 4.1. The *trace* field of initial MAETT MT_0 contains an empty list **nil**. As more instructions are fetched, the MAETT adds to the list INST items which represent the fetched instructions. For example, the *trace* field of MT_1 and MT_2 store list (i_0^1) and $(i_0^2\ i_1^2)$, respectively.

The *init-ISA* field of a MAETT stores the initial ISA state before the execution of the first instruction in the program. Additionally, the *pre-ISA* and *post-ISA*

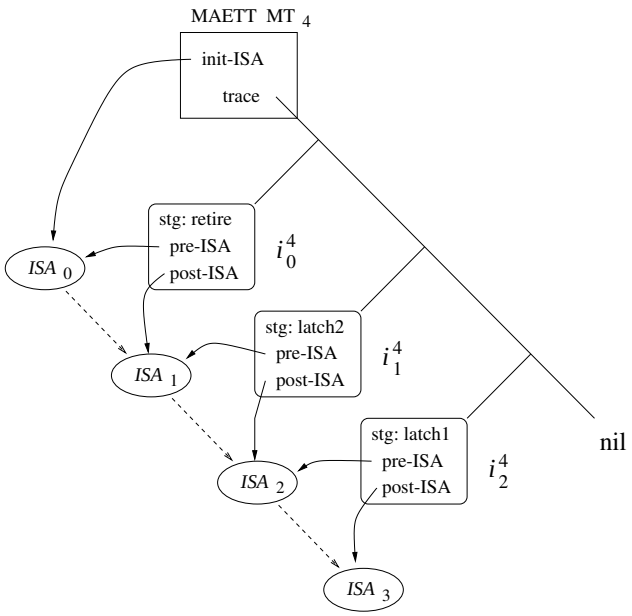

Figure 4.4: The structure of the MAETT intermediate abstraction.

fields of the INST structure store the ISA state before and after executing the represented instruction in the ISA model. We call these states the *pre-ISA state* and the *post-ISA state* of the instruction. Figure 4.4 shows the entire structure of the MAETT MT_4. The *trace* field stores the list $(i_0^4\ i_1^4\ i_2^4)$. The *init-ISA* field stores the initial ISA state ISA_0. This is also the pre-ISA state of the first instruction i_0. The result of executing i_0 is ISA_1 and it is the *post-ISA* state of i_0. Since it is the state before executing the next instruction, ISA_1 is the *pre-ISA* state of i_1. In this way, the MAETT stores all ISA states that appear during the ISA execution of the program. The dashed lines in the figure show the ISA state transitions.

We can define many values related to an instruction using its INST representation. For example, the function INST-word(i_k^t) defines the instruction word of i_k as the memory value addressed by the program counter in the pre-ISA state of i_k. In the following definition, function read-mem(a, mem) defines the value of the memory mem at address a.

DEFINITION:
$\text{INST-word}(i) \stackrel{def}{=} \text{read-mem}((i.\text{pre-ISA}).\text{pc}, (i.\text{pre-ISA}).\text{mem})$

From the instruction word, we can calculate the values in the instruction fields *op*, *ra*, *rb*, and *rc*.

DEFINITION:
$\text{INST-op}(i) \stackrel{def}{=} \text{op-field}(\text{INST-word}(i))$

DEFINITION:
$\text{INST-ra}(i) \stackrel{def}{=} \text{ra-field}(\text{INST-word}(i))$

DEFINITION:
$\text{INST-rb}(i) \stackrel{def}{=} \text{rb-field}(\text{INST-word}(i))$

DEFINITION:
$\text{INST-rc}(i) \stackrel{def}{=} \text{rc-field}(\text{INST-word}(i))$

We can further define the correct source operand values by reading the source registers in the pre-ISA state. Function read-reg(r, regs) returns the value of register r in the register file *regs*.

DEFINITION:
$\text{INST-ra-val}(i) \stackrel{def}{=} \text{read-reg}(\text{INST-ra}(i), (i.\text{pre-ISA}).\text{regs})$

DEFINITION:
$\text{INST-rb-val}(i) \stackrel{def}{=} \text{read-reg}(\text{INST-rb}(i), (i.\text{pre-ISA}).\text{regs})$

Finally, we define INST-result which calculates the execution result of an instruction. The function ALU-output(*op*, *src1*, *src2*) returns the value from the output port of the ALU when the opcode *op*, and source operand values *src1* and *src2* are given to the input ports of the ALU.

DEFINITION:
$\text{INST-result}(i)$
$\stackrel{def}{=}$
$\text{ALU-output}(\text{INST-op}(i), \text{INST-ra-val}(i), \text{INST-rb-val}(i))$

These functions are used in the definition of properties that the pipelined machine should satisfy. For example, predicate INST-latch1-inv defines the correct intermediate values stored in latch1.

DEFINITION:
INST-latch1-inv(i, MA)
$\stackrel{def}{=}$
$\quad$ (((*MA*.latch1).valid?) = 1)
$\wedge$ (((*MA*.latch1).op) = INST-op(i))
$\wedge$ (((*MA*.latch1).rc) = INST-rc(i))
$\wedge$ (((*MA*.latch1).ra) = INST-ra(i))
$\wedge$ (((*MA*.latch1).rb) = INST-rb(i))

We assume that INST i represents an instruction at latch1 in state MA. The busy flag *valid?* of latch1 should be 1, because the latch is occupied by the instruction represented by i. The opcode of i, which has been defined as INST-op(i), should be stored in the *op* field of latch1. Similarly, the predicate checks whether the correct *rc*, *ra*, and *rb* register designators are stored in the corresponding fields of latch1. Another predicate INST-latch2-inv defines the correct intermediate values for latch2.

Using these functions, we define the predicate INST-invariant(i, MA), which is true if and only if the intermediate values for instruction i are correct in state MA, regardless of the stage of i. We define MT-INST-invariant(MT, MA) as a predicate that checks every INST i recorded in the *trace* field of MAETT MT satisfies the condition INST-invariant(i, MA). Intuitively speaking, MT-INST-invariant(MT, MA) checks all pipeline intermediate values are correct.

DEFINITION:
INST-invariant (i, MA)
$\stackrel{def}{=}$
if (i.stg) = `'latch1` **then** INST-latch1-inv (i, MA)
elseif (i.stg) = `'latch2` **then** INST-latch2-inv (i, MA)
else t
fi

DEFINITION:
trace-INST-invariant (*trace*, *MA*)
$\stackrel{def}{=}$
if endp (*trace*) **then t**
else INST-invariant (car (*trace*), *MA*)
$\wedge$ trace-INST-invariant (cdr (*trace*), *MA*)
fi

DEFINITION:
MT-INST-invariant (*MT*, *MA*) $\stackrel{def}{=}$ trace-INST-invariant (*MT*.trace, *MA*)

Another property regs-match-p(MT, MA) is true if and only if the register file in state MA is correct, that is, the results of all completed instructions are stored in the register file. In other words, the register file appears as if it were in the post-ISA state of the last completed instruction. With the example given in Table 4.1, the register file state at time 5 should be the same as that in the post-ISA state of i_1. This ideal register file state is calculated from the MAETT with function MT-regs(MT).

DEFINITION:
trace-regs (*trace*, *ISA*)
$\stackrel{def}{=}$
if endp (*trace*) **then** *ISA*.regs
elseif (car (*trace*).stg) $\neq$ `'retire` **then** *ISA*.regs
else trace-regs (cdr (*trace*), car (*trace*).post-ISA)
fi

DEFINITION:
MT-regs (*MT*) $\stackrel{def}{=}$ trace-regs (*MT*.trace, *MT*.Init-ISA)

DEFINITION:
regs-match-p (*MT*, *MA*) $\stackrel{def}{=}$ MT-regs (*MT*) = (*MA*.regs)

Like MT-INST-invariant(MT, MA) and regs-match-p(MT, MA), we define other properties of our pipelined machine as predicates of the machine state and its MAETT. The following predicate invariant(MT, MA) is the conjunction of such properties.

DEFINITION:
invariant (MT, MA)
$\stackrel{def}{=}$
    pc-match-p (MT, MA)
$\wedge$ regs-match-p (MT, MA)
$\wedge$ mem-match-p (MT, MA)
$\wedge$ ISA-chain-p (MT)
$\wedge$ MT-INST-invariant (MT, MA)
$\wedge$ MT-contains-all-insts (MT, MA)
$\wedge$ MT-in-order-p (MT)

In order to introduce the following two theorems, we need three additional functions. The function flushed?(MA) returns 1 if and only if MA is a pipeline flushed state. The function init-MT(MA) defines the MAETT for any pipeline flushed state MA. The function MT-step(MT, MA, sig) defines the MAETT for the next MA state given that MA is the current MA state and MT is its MAETT.

THEOREM: invariant-init-MT
(MA-state-p (MA) $\wedge$ (flushed? (MA) = 1)) $\rightarrow$ invariant (init-MT (MA), MA)

THEOREM: invariant-step
    (invariant (MT, MA) $\wedge$ MAETT-p (MT) $\wedge$ MA-state-p (MA) $\wedge$ MA-sig-p (sig))
$\rightarrow$ invariant (MT-step (MT, MA, sig), MA-step (MA, sig))

Theorem invariant-init-MT states that every pipeline flushed state satisfies invariant(MT, MA). Theorem invariant-step states that, if invariant(MT, MA) is true for the current state, it is also true for the next state. These two theorems show that property invariant(MT, MA) is an *invariant* condition, and all machine states reachable from a pipeline flushed state satisfy it. In the next section, we prove our commutative diagram using this fact.

4.3 Proving the Commutative Diagram

First we introduce the theorem that we would like to prove. The following theorem is the formal statement of the commutative diagram discussed earlier.

THEOREM: commutative-diagram

$$
\begin{array}{ll}
 & (\quad \text{MA-state-p}\,(MA) \\
 & \quad \wedge\ \text{MA-sig-listp}\,(\textit{sig-list}) \\
 & \quad \wedge\ (n \leq \text{len}\,(\textit{sig-list})) \\
 & \quad \wedge\ (\text{flushed?}\,(MA) = \mathtt{1}) \\
 & \quad \wedge\ (\text{flushed?}\,(\text{MA-stepn}\,(MA,\ \textit{sig-list},\ n)) = \mathtt{1})) \\
\rightarrow & (\quad \text{proj}\,(\text{MA-stepn}\,(MA,\ \textit{sig-list},\ n)) \\
 & \quad = \text{ISA-stepn}\,(\text{proj}\,(MA),\ \text{num-insts}\,(MA,\ \textit{sig-list},\ n)))
\end{array}
$$

In this theorem, MA is the initial state from which the MA execution starts, and it corresponds to MA_0 in Fig 4.3. We consider the execution of n-steps with the list of input signals *sig-list*. The length of *sig-list* should be larger than or equal to n. The result of n-step execution is given as MA-stepn$(MA, \textit{sig-list}, n)$, which corresponds to MA_n in the figure. Suppose the initial state MA and the final state MA-stepn$(MA, \textit{sig-list}, n)$ are both pipeline flushed states. The equality in the conclusion compares the two paths of the commutative diagram. The left-hand side runs the MA machine for n-steps and projects the result to the final ISA state. The right-hand side first projects MA to the initial ISA state proj(MA) and then runs the ISA machine for num-insts$(MA, \textit{sig-list}, n)$ steps. The function num-insts returns the number of instructions executed in the n-step MA execution, which is given as m in Fig. 4.3.

One question is how the function num-insts counts the number of instructions executed during the n-step MA execution. The function num-insts$(MA_0, \textit{sig-list}, n)$ first constructs the MAETT for the final MA state MA_n. This MAETT is a complete history of instructions executed during the n-step MA execution. The function num-insts simply counts the number of the instructions recorded in this MAETT.

DEFINITION:

$$\text{MT-num-insts}\,(MT) \stackrel{def}{=} \text{len}\,(MT.\text{trace})$$

DEFINITION:

$$
\begin{array}{l}
\text{num-insts}\,(MA,\ \textit{sig-list},\ n) \\
\stackrel{def}{=} \\
\text{MT-num-insts}\,(\text{MT-stepn}\,(\text{init-MT}\,(MA),\ MA,\ \textit{sig-list},\ n))
\end{array}
$$

A proof sketch of THEOREM commutative-diagram follows. Component by component, we show the equality in the theorem. There are three components to compare: the program counter, the register file, and the memory. We will discuss the equality with respect to the register file in detail. Equalities for other components are proven similarly.

To ease the following arguments, we use the symbols shown in Fig. 4.3. The left-hand side of the conclusion of THEOREM commutative-diagram is given as proj(MA_n). Since the initial ISA state proj(MA) is ISA_0 in the figure, the right-hand side is given as ISA-stepn(ISA_0, m). We need to prove the following equality for the register file:

$$(\text{proj}(MA_n)).\text{regs} = (\text{ISA-stepn}(ISA_0, m)).\text{regs} \ . \tag{4.1}$$

Let us assume that MT_n represents the MAETT for state MA_n. From the two lemmas invariant-init-MT and invariant-step, invariant(MT_n, MA_n) is true. With the definition of invariant(MT, MA), the property regs-match-p(MT_n, MA_n) is derived. The definition of regs-match-p implies that the final register file state MA_n.regs is equal to the ideal register file state MT-regs(MT_n). Using the definition of proj,

$$(\text{proj}(MA_n)).\text{regs} = MA_n.\text{regs} = \text{MT-regs}(MT_n) \ .$$

Let ($i^n_0 \ \ldots \ i^n_{m-1}$) be the list of instructions in the *trace* field of MAETT MT_n. Because the final state MA_n is flushed, the execution of all instructions in this list are completed and i^n_k.stg = `'retire` for all k such that $0 \leq k < m$. Hence, MT-regs(MT_n) = (i^n_{m-1}.post-ISA).regs because MT-regs(MT_n) returns the register file in the post-ISA state of the last completed instruction, which is represented by i^n_{m-1}. Since the post-ISA state of i^n_{m-1} is the state that results from executing m instructions i_0 through i_{m-1} by the ISA, it is equal to ISA-stepn(ISA_0, m). Therefore,

$$
\begin{aligned}
&\text{MT-regs}(MT_n) \\
&= \text{INST-post-ISA}(i^n_{m-1}).\text{regs} \\
&= \text{ISA-stepn}(ISA_0, m).\text{regs}\ .
\end{aligned}
$$

From the equalities shown above, we derive Formula (4.1) and conclude the proof of the commutative diagram with respect to the register file.

The THEOREM commutative-diagram is *vacuous* if the MA never reaches a pipeline flushed state, because the last hypothesis of the theorem does not hold. However, the following theorem proves that we can flush the pipelined machine by running the MA model long enough without fetching new instructions. The input signal to the pipelined machine controls instruction fetching, and the machine does not fetch a new instruction when the input is 0.

THEOREM: liveness
  MA-state-p (MA)
$\rightarrow$ (flushed? (MA-stepn (MA, zeros (flush-cycles (MA)), flush-cycles (MA))) = 1)

Function zero(n) returns a list of 0's whose length is n. The *witness function* flush-cycles(MA) returns the number of steps necessary to flush out all instructions in the pipeline, proving the existence of such a number.

Even though the 3-stage pipelined machine verified here is simple, our verification approach can be scaled to a more complex pipelined machine model. We later use a similar approach to verify a microprocessor model which issues and completes instructions out-of-order, executes instructions speculatively, and implements interrupts. To verify such a processor model, we had to extend the MAETT to record more information about instructions. Also we needed to verify more complex invariants than the 3-stage pipelined machine. However, the general approach to the problem does not change.

Chapter 5

Machine Specification of the FM9801

The FM9801 microprocessor is a new microprocessor model we invented for our research project[SH98]. The verification of this microprocessor design is the main topic of this dissertation. The FM9801 implements various microprocessor design techniques found in modern microprocessors such as speculative execution, precise exceptions, and out-of-order issue and completion of instructions. The FM9801 microprocessor model is not intended to be fabricated, nor it is not as complicated as industrial microprocessors. Still it is not a toy example, but a realistic model which reveals verification problems that may not be foreseen by verifying simplified models.

We formally specify the FM9801 microprocessor at the *instruction-set architecture* (ISA) level and the *microarchitecture* (MA) level. The ISA model contains only the components visible to the programmer, such as the program counter, the register file and the memory. The ISA defines how individual instructions modify the states of programmer visible components. The ISA model executes instructions sequentially, completing one instruction before starting another. On the other hand,

the MA model is a clock-cycle-accurate model of a pipelined machine implementation; the state transition of the MA model corresponds to a hardware clock cycle. The MA model contains all microarchitectural components, regardless of their visibility to the programmer. The ISA model is our machine specification, and the MA model is our verification target.

We first discuss the basic components of the FM9801 in Section 5.1. We then discuss the ISA model of the FM9801 in Section 5.2. Finally, we explain its MA model in Section 5.3.

5.1 Basic Components of the FM9801

In this section, we discuss the program counter, the general-purpose register file, the special register file, and the memory in the FM9801. These are the components visible to the programmer and they are included in both the ISA and the MA.

5.1.1 Address and Data Word

The size of the memory space for the FM9801 is 2^{16}, and a 16-bit *address word* can address the entire memory. The memory at each address contains a 16-bit instruction and data word, not a 8-bit byte. A similar memory architecture is used in the design of FM8501 and FM9001 [Hun94, BHK94].

With the IHS library, we represent a 16-bit address word with an integer between 0 and $2^{16} - 1$. The type predicate, addr-p, and type coercion function, addr, are defined by the **defbytetype** macro in the IHS library to satisfy the following theorems:

THEOREM: addr-p-type-def
addr-p $(x) \leftrightarrow ((x \in \textbf{Int}) \wedge (0 \leq x) \wedge (x < \text{expt}(2, 16)))$

THEOREM: addr-mod
$(x \in \textbf{Int}) \rightarrow (\text{addr}(x) = (x \textbf{ mod } \text{expt}(2, 16)))$

The type coercion function addr(x) converts x to an integer representing a 16-bit address word.

Data words in the register and the memory of the FM9801 are 16-bit. The IHS **defbytetype** macro defines the type predicate, word-p, and the type coercion function, word, to satisfy the following:

THEOREM: word-p-type-def
word-p $(x) \leftrightarrow ((x \in \mathbf{Int}) \wedge (0 \leq x) \wedge (x < \text{expt}\,(2, 16)))$

THEOREM: word-mod
$(x \in \mathbf{Int}) \rightarrow (\text{word}\,(x) = (x \ \mathbf{mod}\ \text{expt}\,(2, 16)))$

5.1.2 Program counter

The program counter stores the address from which the next instruction is fetched. When an instruction is fetched, the program counter is incremented modulo 2^{16}. If pc is the current program counter value, the new program counter value is expressed as addr($pc + 1$). Thus, if the current program counter value is $2^{16} - 1$, the program counter is set to 0 after fetching an instruction.

5.1.3 Register Files

The FM9801 has two register files: one for general-purpose registers, and the other for special registers. We may call a general-purpose register simply a register in this dissertation. General-purpose registers hold the data used in normal program executions. Special registers store the information related to exceptions and the processor's privilege mode.

The FM9801 has 16 general-purpose registers, each of which stores a 16-bit data word. We represent a general-purpose register file state with a list of 16 integers representing data words. The type predicate RF-p(RF) is true if RF is a list representing a general-purpose register file state.

The register accesses to the general-purpose register file are defined by two functions read-reg(r, RF) and write-reg(v, r, RF). The function read-reg(r, RF) returns the value of the register designated by r in register file RF. The register designator r satisfies type predicate rname-p(r), whose definition is shown in Appendix B.1. The function write-reg(v, r, RF) defines the state of register file RF after the register designated by r is modified with the new value v. We define read-reg and write-reg as:

DEFINITION:
read-reg $(r,\ RF) \stackrel{def}{=}$ nth $(r,\ RF)$

DEFINITION:
write-reg $(val,\ r,\ RF) \stackrel{def}{=}$ update-nth $(r,\ val,\ RF)$

where nth(n, lst) returns the n'th element of list lst, and update-nth(v, n, lst) returns list lst after replacing the n'th element with v. The functions read-reg and write-reg satisfy the following lemma.

THEOREM: read-reg-write-reg
(rname-p $(r1)$ $\wedge$ rname-p $(r2)$ $\wedge$ RF-p (RF))
$\rightarrow$ (read-reg $(r1,$ write-reg $(val,\ r2,\ RF))$
$=$ **if** $r1 = r2$ **then** val
else read-reg $(r1,\ RF)$
fi)

This theorem shows that write-reg$(v, r2, RF)$ modifies the register designated by $r2$ and keeps other registers unchanged.

The special register file contains two 16-bit special registers and a flag to specify the privilege mode. A special register file state is defined as a structure SRF$(su, sr0, sr1)$ in Appendix B.2, whose type predicate is SRF-p(SRF). The field su is the 1-bit flag indicating the privilege mode, $sr0$ is special register 0, and $sr1$ is special register 1. When the flag su is 1, the processor is in *supervisor mode*. Otherwise, the processor is in *user mode*. Privileged instructions can be safely executed only in supervisor mode. Execution of a privileged instruction in

user mode raises an illegal instruction exception. The memory protection is also enforced only in supervisor mode.

Read and write accesses to the special registers 0 and 1 are defined by the functions read-sreg(sr, SRF) and write-sreg(v, sr, SRF), where sr is the special register designator satisfying srname-p(sr). They satisfy the following theorem:

THEOREM: read-sreg-write-sreg
  (srname-p (*r1*) ∧ srname-p (*r2*) ∧ SRF-p (*SRF*))
→ (read-sreg (*r1*, write-sreg (*val*, *r2*, *SRF*))
  = **if** *r1* = *r2* **then** *val*
    else read-sreg (*r1*, *SRF*)
    fi)

5.1.4 Memory

The FM9801 has a 2^{16} bit address space. A memory read access is defined by a function read-mem(a, mem), which returns the 16-bit data word stored in the memory mem at address a. The function write-mem(v, a, mem) defines the memory state after modifying the memory mem at address a with data word v. Functions read-mem and write-mem satisfy the following theorem:

THEOREM: read-mem-write-mem
  (addr-p (*ad1*) ∧ addr-p (*ad2*) ∧ mem-p (*mem*))
→ (read-mem (*ad1*, write-mem (*val*, *ad2*, *mem*))
  = **if** *ad1* = *ad2* **then** *val*
    else read-mem (*ad1*, *mem*)
    fi)

The FM9801 memory system has page-wise memory protection. A memory page consists of 2^{10} words and has its own protection mode. There are three memory protection modes: `'no-access`, `'read-only`, and `'read-write`. If a page is in the `'no-access` mode, neither read accesses nor write accesses are allowed to any memory words in the page in user mode. If the page is in the `'read-only` mode, only read accesses are allowed. If it is in the `'read-write` mode, both read and write accesses are allowed. The function readable-addr?(a, mem) returns 1 iff the

address a is readable in memory state mem. The function writable-addr?(a, mem) returns 1 iff the address a is writable. The FM9801 enforces the memory protection only when the processor is in user mode.

5.1.5 Efficient Memory Model

The formal specification of the FM9801 memory must provide a well-defined semantics of the memory accesses for verification. The execution speed of the formal specification is also important because we use the same specification for simulation purposes. In this subsection, we discuss a number of approaches to formally specify a memory system, and explain our approach to the problem.

Representing the entire memory with a linear linked list or a linear array is not a realistic solution for simulations because it consumes a huge memory space in the simulating machine. Although the FM9801 memory space is relatively small with only 2^{16} data words, memory systems with a 32-bit or 64-bit address space are more common today. It is also difficult to define the formal semantics of destructive accesses to an array in an applicative functional language like the ACL2 logic

One approach to the compact representation of memory states with a well-defined semantics uses association lists. A tuple $(a\ .\ v)$ represents a memory value v stored at an address a. The entire memory state is represented as a list of such association tuples. If $(a\ .\ v)$ is the first tuple in the list whose first element is a, we interpret the memory value at address a to be v. If no tuple in the list has a as its first element, we interpret that the memory at address a stores the default value. This approach can concisely represent the entire memory space by recording the values in only the modified portion of the memory. The semantics of the model in an applicative functional language is straightforwardly defined. However, simulating a memory read access may have to scan the entire list. A memory write operation is simply appending a tuple of the accessed address and the new value at the head of

the list. However, the association list grows as more write operations are simulated, making the simulation of read operations increasingly time-consuming.

The ACL2 theorem prover system provides the ACL2 array facility. The idea is using an association list representation to provide a formal semantics of array accesses, while allowing an efficient simulation using a real array. The association list and the array are tied together "under the hood", and the implementation details are hidden from a user. Consequently, the time required to simulate read and write accesses to an array is constant. However, defining the entire memory space as an ACL2 array is not practical because the ACL2 array allocates the entire space as a real array during the simulation.

A binary tree representation of the memory can be a good compromise between the simulation performance and the space requirement [Yu90, BHK94]. A binary tree can concisely represent the entire memory address space by dynamically allocating a tree node corresponding to a memory address when the first access to the address is simulated. A single memory access can be simulated in the time logarithmic to the size of the simulated memory address space. However, in the simulation of the memory with a 16-bit address space, a memory access needs to traverse 16 nodes in the binary tree, making the execution speed significantly slower than the linear array representation of a memory model.

We implement the FM9801 memory model using a hierarchical data-structure using the ACL2 arrays. Figure 5.1 illustrates the data-structure with two levels. The array at the upper level contains a page entry at each index. Each page entry contains a pointer to an array at the lower level. Each array at the lower level records 2^{10} words in the represented page. The lower level array is allocated dynamically when the first access to the corresponding page is simulated. Since all arrays in the memory models are ACL2 arrays, the semantics of the memory model is well-defined in the ACL2 logic.

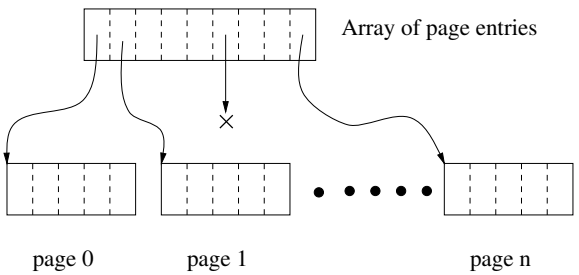

Figure 5.1: Data Structure for Memory Model

By using the two level data-structure and allocating the second level arrays on demand, the space required to simulate a memory system is proportional to the number of actually used pages. The time required to simulate a memory access is constant and comparable to the linear array representation of the memory, because we need only two array references to simulate each memory access.

5.2 Instruction-Set Architecture of the FM9801

The instruction-set architecture (ISA) defines the behavior of the machine from the programmer's viewpoint by specifying the effect of individual instructions. The ISA executes instructions sequentially, completely executing one instruction before starting another. The ISA plays the role of the specification as opposed to the microarchitectural design which is our implementation.

5.2.1 Instruction-Set Architecture State

An ISA state consists of the states of programmer visible components. An ISA state is represented as ISA-state(pc, rf, srf, mem) where pc, rf, srf, and mem represent the states of the program counter, the general-purpose register file, the special register file, and the memory, respectively. See Appendix B.3 for the formal definition of the data structure used to represent ISA states.

5.2.2 The Instruction Set of the FM9801

The FM9801 implements 11 instruction types. We could have implemented more instructions without adding too much complexity to the machine design. For example, additional integer instructions could have been added. Their behaviors in the pipeline can be exactly the same as that of an add instruction, except that they perform different arithmetic operations on their operand values. This only increases the complexity of the arithmetic logic unit; however, it hardly complicates the control logic of the implemented machine. Instead, we have implemented a small number of instruction types, whose behaviors are significantly different from each other in the microarchitectural implementation of the FM9801. Each instruction can be considered as a representative of similar instruction types, like an addition instruction is a representative of all integer instructions with similar control complexity.

We have implemented instruction types that seem interesting from the perspective of the verification of pipelined machines. The FM9801 instruction set includes a conditional branch instruction, memory load and store instructions, privileged instructions, and instructions that synchronize the pipeline or that change the privilege mode of the processor.

The FM9801 uses a 16-bit data word to represent an instruction. There are four instruction formats, A, B, C, and D, as shown in Fig 5.2. The field *opcode* specifies the instruction type. The fields *ra*, *rb*, and *rc* designate operand registers. The field *im* stores an immediate value.

The instructions implemented in the FM9801 are listed in Table 5.1. An instruction word whose opcode field value is not shown in the table is an illegal instruction. Some instructions are privileged. Privileged instructions can be executed only when the processor is in supervisor mode.

In the rest of this section, we explain the semantics of each instruction. The effect of normal execution of an instruction is provided as a sequence of assignments.

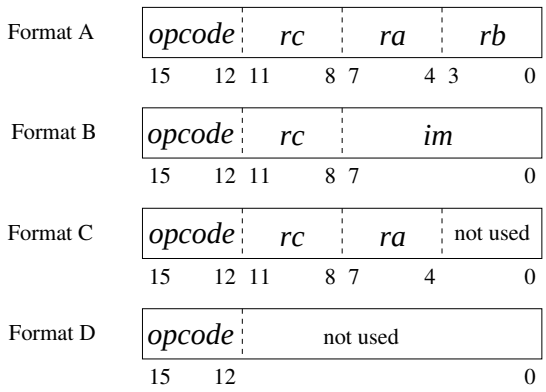

Figure 5.2: Instruction format A, B, C, and D for the FM9801

Mnemonic	opcode	Format	Privileged	Semantics
`ADD`	0	A	No	Add
`MUL`	1	A	No	Multiply
`BR`	2	B	No	Conditional Branch
`LD`	3	A	No	Load
`ST`	4	A	No	Store
`SYNC`	5	D	No	Synchronize
`LDI`	6	B	No	Load with Immediate Address
`STI`	7	B	No	Store with Immediate Address
`RFEH`	8	D	Yes	Return from Exception Handling
`MFSR`	9	C	Yes	Move Word from Special Register
`MTSR`	10	C	Yes	Move Word to Special Register

Table 5.1: FM9801 Instruction Set

In these assignments, pc, RF, SRF and mem represent the current states of the program counter, the general-purpose register file, the special register file, and the memory. Primed variables, pc', RF', SRF' and mem' represent the new states of the corresponding components after the instruction is executed. The effects of exceptions and interrupts are discussed later.

Addition — ADD *rc,ra,rb*

$$\begin{aligned} val &\leftarrow \text{word}(\text{read-reg}(ra, RF) + \text{read-reg}(rb, RF)) \\ RF' &\leftarrow \text{write-reg}(val, rc, RF) \\ pc' &\leftarrow \text{addr}(pc + 1) \end{aligned}$$

An ADD instruction adds the values of the registers designated by the *ra* and *rb* fields and stores the result into the register designated by the *rc* field. If the result overflows, the least significant 16 bits of the sum are stored in the register *rc*.

Multiply — MUL *rc,ra,rb*

$$\begin{aligned} val &\leftarrow \text{word}(\text{read-reg}(ra, RF) \times \text{read-reg}(rb, RF)) \\ RF' &\leftarrow \text{write-reg}(val, rc, RF) \\ pc' &\leftarrow \text{addr}(pc + 1) \end{aligned}$$

The values of the *ra* and *rb* registers are multiplied and the result is stored in the *rc* register. If the result overflows, the least significant 16 bits are used as the result.

Conditional Branch — BR *rc,im*

$$\begin{aligned} &\text{If read-reg}(rc, RF) = 0 \quad \text{then} \quad pc' \leftarrow \text{addr}(pc + \text{logextu}(8, 16, im)), \\ &\text{otherwise,} \qquad\qquad\qquad\qquad\quad pc' \leftarrow \text{addr}(pc + 1). \end{aligned}$$

A conditional branch is taken when the *rc* register is 0. The branch target address is calculated by sign-extending the 8-bit immediate value im to 16-bits, adding it to the old program counter value, and taking the modulo 2^{16} of the result. The function $\text{logextu}(n, m, x)$ interprets integer x as a n-bit bit vector and sign-extends it to m bits. When the value of the *rc* register is equal to 0, the branch is taken and

the branch target is stored in the program counter. Otherwise, the branch is not taken; the program counter is set to the wrap-around increment of the old program counter value.

Load — LD *rc*,*ra*,*rb*

$$\begin{aligned} ad &\leftarrow \text{addr}(\text{read-reg}(ra, RF) + \text{read-reg}(rb, RF)) \\ val &\leftarrow \text{read-mem}(ad, mem) \\ RF' &\leftarrow \text{write-reg}(val, rc, RF) \\ pc' &\leftarrow \text{addr}(pc + 1) \end{aligned}$$

A data word is read from the memory and stored in the rc register. The memory access address is the wrap-around sum of the ra and rb register values. If an LD instruction is executed in user mode and if the access address is read-protected, a data access error exception occurs. The exception does not occur if the processor is in supervisor mode.

Store — ST *rc*,*ra*,*rb*

$$\begin{aligned} ad &\leftarrow \text{addr}(\text{read-reg}(ra, RF) + \text{read-reg}(rb, RF)) \\ val &\leftarrow \text{read-reg}(rc, RF) \\ mem' &\leftarrow \text{write-mem}(val, ad, mem) \\ pc' &\leftarrow \text{addr}(pc + 1) \end{aligned}$$

The value of the rc register is stored in the memory at the access address, which is the wrap-around sum of the ra and rb register values. If an ST instruction is executed in user mode and if the access address is write-protected, a data-access error exception occurs. The exception does not occur if the processor is in supervisor mode.

Synchronization — SYNC

$$pc' \leftarrow \text{addr}(pc + 1)$$

This instruction can be used to serialize the execution of instructions. No state changes occur on programmer visible components except that the program counter

is incremented. However, the pipelined implementation of the FM9801 will flush the instructions in the pipeline and synchronize the machine when the instruction is executed. This is useful when explicit serialization is necessary. For example, self-modifying code can be safely executed by first executing the instructions that modify the program, synchronizing the machine, and then executing the modified instructions.

Load with Immediate Address — LDI *rc*,*im*

$$\begin{aligned} ad &\leftarrow \text{addr}(im) \\ val &\leftarrow \text{read-mem}(ad, mem) \\ RF' &\leftarrow \text{write-reg}(val, rc, RF) \\ pc' &\leftarrow \text{addr}(pc + 1) \end{aligned}$$

The memory access address is calculated by unsigned-extending the 8-bit immediate value *im* to 16-bits. The memory value at the access address is stored in the *rc* register. If an LDI instruction is executed in user mode and the access address is read-protected, then a data access error exception occurs.

Store with Immediate Address — STI *rc*,*im*

$$\begin{aligned} ad &\leftarrow \text{addr}(im) \\ val &\leftarrow \text{read-reg}(rc, RF) \\ mem' &\leftarrow \text{write-mem}(val, ad, mem) \\ pc' &\leftarrow \text{addr}(pc + 1) \end{aligned}$$

The memory access address is calculated by unsigned-extending the 8-bit immediate value *im* to 16-bits. The *rc* register value is stored in the memory at the access address. If an STI instruction is executed in user mode and the access address is write-protected, a data access error exception occurs.

Return From Exception Handler — RFEH

$$
\begin{aligned}
sr0 &\leftarrow \text{read-sreg}(0, SRF) \\
sr1 &\leftarrow \text{read-sreg}(1, SRF) \\
su' &\leftarrow \text{logcar}(sr1) \\
SRF' &\leftarrow \text{SRF}(su', sr0, sr1) \\
pc' &\leftarrow \text{addr}(sr0)
\end{aligned}
$$

This instruction is the only way to switch from supervisor mode to user mode. The transition from user mode to supervisor mode only occurs when an exception or an interrupt is processed. A typical use of this instruction is to return from an exception handler to the interrupted user program. The least significant bit of special register 1 is used as the new value of the *su* flag. In other words, special register 1 specifies the privilege mode after the RFEH instruction is executed. The program counter is set to the value of the special register 0. An RFEH instruction synchronizes the machine so that the subsequent instructions are executed in the correct privilege mode. RFEH is a privileged instruction. More discussions about exception handling can be found in Subsection 5.2.3.

Move from Special Register — MFSR *rc*,*ra*

$$
\begin{aligned}
val &\leftarrow \text{read-sreg}(ra, SRF) \\
RF' &\leftarrow \text{write-reg}(val, rc, RF) \\
pc' &\leftarrow \text{addr}(pc + 1)
\end{aligned}
$$

The value of the special register designated by the *ra* field is stored in the general-purpose register designated by the *rc* field. Legitimate *ra* field values are 0 and 1. Otherwise, *ra* does not designate an existing special register, and an illegal instruction exception occurs. MFSR is a privileged instruction.

Move to Special Register — MTSR rc,ra

$$\begin{aligned} val &\leftarrow \text{read-reg}(rc, RF) \\ SRF' &\leftarrow \text{write-sreg}(val, ra, SRF) \\ pc' &\leftarrow \text{addr}(pc + 1) \end{aligned}$$

The value of the general-purpose register designated by the rc field is stored in the special register designated by the ra field. Legitimate ra field values are 0 and 1. Otherwise, an illegal instruction exception occurs. MTSR is a privileged instruction.

5.2.3 Exceptions and Interrupts in the FM9801

The FM9801 implements *exceptions* and *interrupts*. When an exception or an interrupt is detected, the program execution is suspended and the processor starts the execution of another program called the *exception handler*.

We introduce four types of exception and interrupts. Sometimes the terms "interrupts" and "exceptions" are used interchangeably. "There is no accepted nomenclature associated with interrupts; every manufacturer seems to be creative in their use of terms." [Cra96] In this dissertation, we classify exceptions and interrupts in the FM9801 as follows:

- **External Interrupt** or simply interrupt. External interrupts are caused by signals to the microprocessor. Upon receiving an external interrupt signal, the FM9801 interrupts the currently executing program, and starts the execution of the exception handler. The program is interrupted asynchronously; any instruction can be interrupted by an interrupt signal.

- **Internal Exception** or simply exception. When the microprocessor detects an error during the execution of an instruction, the program execution is interrupted at the instruction and the exception handler is executed. The FM9801 implements three internal exceptions:

- **Fetch Error Exception** If the processor attempts to fetch an instruction from a read-protected portion of the memory in user mode, a fetch error exception is raised.
- **Illegal Instruction Exception** If an instruction has a non-defined opcode, an illegal instruction exception is raised. An illegal instruction is also detected if the processor attempts to execute a privileged instruction in user mode, or the instructions specify non-existing special registers as operands.
- **Data Access Error Exception** If a memory-load instruction attempts to read from a read-protected portion of the memory in user mode, a data access error exception is raised. Similarly, if a memory-store instruction attempts to write to a write-protected portion of the memory, a data access error exception is raised.

No exceptions and interrupts are maskable, i.e., there is no way to disable exceptions and interrupts. The highest priority is given to the external interrupt, followed by the fetch error, the illegal instruction, and the data-access error in that order.

Assume, in the ISA state before an exception is detected, that the values of the program counter, the privilege mode flag, special register 0, and special register 1 are given as pc, su, $sr0$, and $sr1$, respectively. The states of these components at the beginning of exception handling are represented as pc', su', $sr0'$, and $sr1'$ below.

$$
\begin{aligned}
pc &\leftarrow < \textit{Exception Handler Address} > \\
su' &\leftarrow 1 \\
sr0' &\leftarrow \begin{cases} \text{word}(pc+1) & \text{if execution resumes at the next instruction} \\ \text{word}(pc) & \text{if execution resumes at the current instruction} \end{cases} \\
sr1' &\leftarrow \text{word}(su)
\end{aligned}
$$

Exception Type	Exception Handler Address (Hex)	Resume Execution From
External Interrupt	0030	current instruction
Fetch Error	0010	current instruction
Illegal Instruction	0000	next instruction
Data Access Error	0020	current instruction

Table 5.2: Exception and Interrupts in the FM9801

When an exception is detected, the program counter is set to the first instruction of the exception handler, whose address is shown in Table 5.2 for each exception type. The privilege mode is set to supervisor mode. The old state of the privilege mode flag, *su*, is saved in the special register 1. The old program counter value is saved in special register 0. The program counter value is incremented before it is saved, if the interrupted program is supposed to resume its execution from the next instruction. Table 5.2 shows whether the program should resume its execution from the interrupted instruction or the next instruction depending on the exception type. For example, a resumed program does not execute the illegal instruction that has caused an exception.

When the exception handling completes, an RFEH instruction is used to resume the execution of the interrupted program. The effect of the RFEH instruction was discussed in the previous subsection. Upon calling RFEH, the exception handler should set the special register 0 to the return address, and specify the new privilege mode by setting the least significant bit of the special register 1. If the exception handler has not changed both special registers after the exception is raised, the execution of the RFEH instruction restores the original privilege mode and the program counter value.

5.2.4 Formal Definition of the ISA

We formally define the FM9801 ISA in the ACL2 logic. Using a Lisp-like language such as the ACL2 logic enables us to define an executable instruction-set specification [Cra83]. Not only we can use our ISA definition as the specification of the MA implementation during the formal verification, but it can also be used for simulation. A programmer can also refer to the ISA specification as a "formula manual" of the microprocessor [HS99].

The next ISA state function ISA-step(ISA, $intr$) specifies the behavior of the ISA. This function takes the current state ISA and external interrupt signal $intr$ and returns the ISA state after executing one instruction. It is natural to define the ISA-step by specifying the effect of individual instructions.

The effect of each instruction type is defined with the functions shown in Table 5.3. For example, ISA-add(rc, ra, rb, ISA) defines the next ISA state after executing an ADD instruction with operands rc, ra, and rb.

```
DEFINITION:
ISA-add(rc, ra, rb, ISA)
def
=
let* pc be ISA.pc,
     RF be ISA.RF,
     val be word(read-reg(ra, RF) + read-reg(rb, RF))
      in
ISA-state(addr(pc + 1), write-reg(val, rc, RF), ISA.SRF, ISA.mem)
```

We also define four functions to specify the effects of interrupts and exceptions listed in Table 5.4. These functions take as an argument the current ISA state and return the ISA state at the beginning of the execution of the exception handler. For example, ISA-fetch-error(ISA) is defined as follows:

```
DEFINITION:
ISA-fetch-error(ISA)
def
=
ISA-state(10_16, ISA.RF, SRF(1, word(ISA.pc), word(ISA.SRF.su)), ISA.mem)
```

Instruction	Function
ADD	ISA-add(rc, ra, rb, ISA)
MUL	ISA-mul(rc, ra, rb, ISA)
BR	ISA-br(rc, im, ISA)
LD	ISA-ld(rc, ra, rb, ISA)
ST	ISA-st(rc, ra, rb, ISA)
SYNC	ISA-sync(rc, ISA)
LDI	ISA-ldi(rc, im, ISA)
STI	ISA-sti(rc, im, ISA)
RFEH	ISA-rfeh(ISA)
MFSR	ISA-mfsr(rc, ra, ISA)
MTSR	ISA-mtsr(rc, ra, ISA)

Table 5.3: Functions defining the effect of instructions

Exception Type	Function
Fetch Error	ISA-fetch-error(ISA)
Illegal Instruction	ISA-illegal-inst(ISA)
Data Access Error	ISA-data-accs-error(ISA)
External Interrupt	ISA-external-intr(ISA)

Table 5.4: Functions defining the effect of exceptions and interrupts

This function defines the state after a fetch error is detected. The program counter is set to 10_{16}. The privilege mode flag is set to 1, the old program counter value *ISA*.pc is saved in special register 0, and the old privilege mode flag *ISA*.SRF.su is saved in the special register 1.

```
DEFINITION:
ISA-step (ISA, intr)
def
=
if ISA-oracle-exint (intr) = 1 then ISA-external-intr (ISA)
 elseif read-error? (ISA.pc, ISA.mem, ISA.SRF.su) = 1
 then ISA-fetch-error (ISA)
 else let inst be read-mem (ISA.pc, ISA.mem)
          in
      let op be opcode (inst),
          rc be rc (inst),
          ra be ra (inst),
          rb be rb (inst),
          im be im (inst)
          in
      if op = 0 then ISA-add (rc, ra, rb, ISA)
       elseif op = 1 then ISA-mul (rc, ra, rb, ISA)
       elseif op = 2 then ISA-br (rc, im, ISA)
       elseif op = 3 then ISA-ld (rc, ra, rb, ISA)
       elseif op = 6 then ISA-ldi (rc, im, ISA)
       elseif op = 4 then ISA-st (rc, ra, rb, ISA)
       elseif op = 7 then ISA-sti (rc, im, ISA)
       elseif op = 5 then ISA-sync (ISA)
       elseif op = 8 then ISA-rfeh (ISA)
       elseif op = 9 then ISA-mfsr (rc, ra, ISA)
       elseif op = 10 then ISA-mtsr (rc, ra, ISA)
       else ISA-illegal-inst (ISA)
      fi
fi
```

The definition of ISA-step shown above reads an instruction from the memory, determines the type of the instruction from the opcode of the instruction, and calls the corresponding function with operand values. It also specifies the behavior on the interrupts and exceptions. If the argument *intr* is 1, the ISA interrupts the next instruction. The function ISA-stepn(*ISA*, *intr-list*, n) defines the ISA state after an n-step execution with a list of external interrupt signals *intr-list*.

DEFINITION:
ISA-stepn(ISA, *intr-lst*, n)
$\stackrel{def}{=}$
if $n \simeq 0$ **then** ISA
else ISA-stepn(ISA-step(ISA, car(*intr-lst*)), cdr(*intr-lst*), $n-1$)
fi

5.3 Microarchitectural Design of FM9801

The FM9801 specification at the MA level is a clock-cycle accurate model of the microprocessor design. It defines the behavior of all components used in the implementation regardless of their visibility to the programmer.

The block diagram of the FM9801 is shown in Fig. 5.3. When an instruction is fetched, it is initially stored in the *instruction fetch unit* (IFU). The fetched instruction is decoded and sent to the *dispatch queue*. The instruction is then *dispatched* to one of the *reservation stations* where it waits for its operands. When all its operands become ready, the instruction is *issued*[1] to an execution unit. There are four execution units in the FM9801: the integer unit, the multiply unit, the branch unit, and the load-store unit. Each instruction is executed in the execution unit appropriate for the instruction type. When the execution unit *completes* the execution of an instruction, the result is routed through the *common data bus* (CDB) to the *reorder buffer*. The instruction waits in the reorder buffer to be *committed*. It is when the instruction is committed that its result is written to the register file. The results from the execution units are also *forwarded* through the CDB to the reservation stations, where other instructions wait for the results as operands. The data memory access for a store instruction may be pending in the write-buffer in the load-store unit even after the instruction is committed. When all operations related to an instruction complete, the instruction is said to be *retired*. When a

[1]The terms "dispatch" and "issue" are sometimes used differently. Some people regard the reservation station as a virtual execution unit instead of as an instruction window. They call instruction dispatch and issue in our definitions "issue" and "release", respectively.

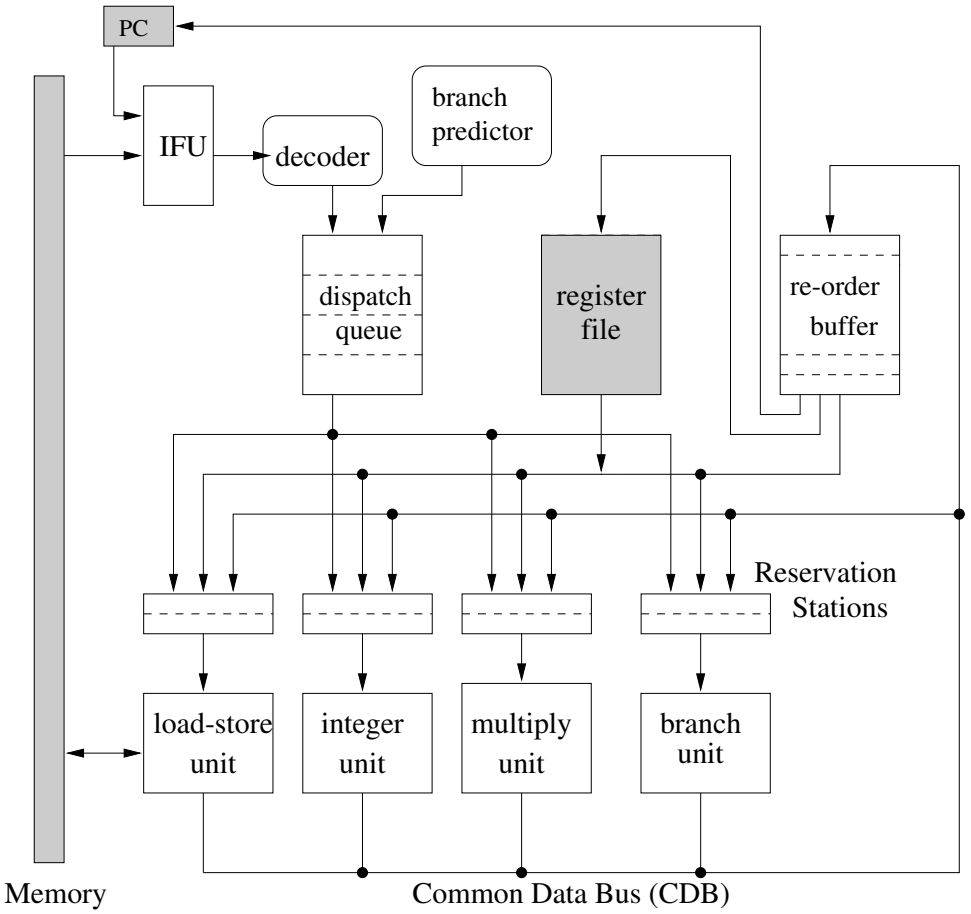

Figure 5.3: Block Diagram of the FM9801.

conditional branch instruction is decoded, a branch predictor predicts whether the branch will be taken. Instructions following a conditional branch are speculatively executed until the branch instruction is committed.

A state of the FM9801 MA model is represented in the ACL2 logic with data-structures defined with the **defstructure** macros. The definition of the data-structure is given in Appendix B.4.

In the following subsections, we will discuss each component of the FM9801 microarchitectural design. Detailed descriptions of design techniques used in the FM9801 can be found in the literature [Joh91, PH96, Cra96].

5.3.1 Instruction Fetch Unit and Dispatch Queue.

The IFU fetches an instruction word from the memory addressed by the program counter. The FM9801 fetches at most one instruction in every clock cycle. Figure 5.4 shows the IFU and the dispatch queue. The fields of the IFU, *valid?*, *excpt*, *pc*, and *word*, store a busy flag, the exception status, the associated program counter value, and the fetched instruction word, respectively. The 3-bit field, *excpt*, represents the exception status for the fetched instruction. The encoding shown in Table 5.5 is used to represent an exception status in several components in the FM9801. For example, if an instruction fetch error is detected by the IFU, the *excpt* field is set to 101_2. The value stored in the *pc* field is the address of the fetched instruction word.

The fetched instruction is decoded next, and it is sent to the dispatch queue. The dispatch queue of the FM9801 works as a *centralized instruction window*, which buffers decoded instructions and constantly supplies them to the execution units. The IFU may not fetch instructions fast enough due to slow memory responses.

The dispatch queue is a first-in-first-out (FIFO) buffer with four entries *DQ0*, *DQ1*, *DQ2*, and *DQ3*. The fields, *valid?*, *pc*, and *excpt*, in dispatch queue entries are similar to those in the IFU. The opcode of the instruction is decoded into a control

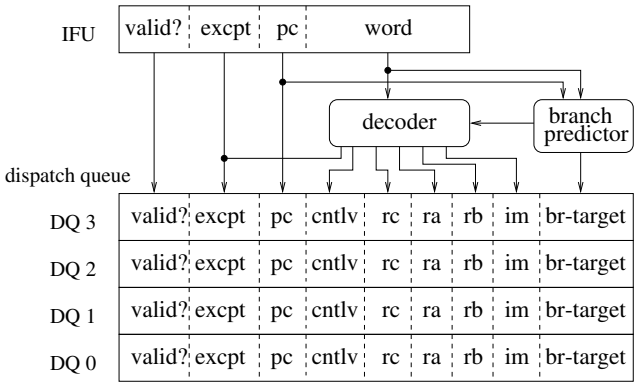

Figure 5.4: The IFU and dispatch queue of the FM9801.

Exception Type	*excpt* field
No Exception Detected	$0XX_2$
Illegal Instruction	100_2
Fetch Error	101_2
Data Access Error	110_2
(External Interrupt)	111_2

Table 5.5: Encoding for the exception flags in the FM9801. X represents a "don't care" bit. Flags for an external interrupt, 111_2, is not used in the current implementation of the FM9801.

vector and stored in the field *cntlv*. The definition of the control vector is given in Table B.3 in the appendix. The dispatch queue fields *ra*, *rb*, *rc*, and *im* store the values from the corresponding instruction fields of the original instruction word.

When a BR instruction is decoded, a branch predictor predicts whether the conditional branch will be taken. A bit of the control vector is used to store the branch prediction result. The branch target address is calculated at the same time and stored into the field *br-target*. We will discuss branch prediction and speculative execution in Subsection 5.3.7 in detail.

5.3.2 Tomasulo's Algorithm

The FM9801 may issue and complete instructions out of the original program order. This is called *out-of-order issue* and *out-of-order completion* of instructions. On the other hand, the instructions are fetched, dispatched and committed *in-order*; that is, the FM9801 fetches, dispatches and commits instructions exactly in program order. We use *Tomasulo's algorithm* [Tom67] to implement out-of-order issue and completion. This algorithm keeps track of the dependencies between instructions and forwards the results of instructions to other instructions that will use previous results as operands.

The key technique in Tomasulo's algorithm is associating a tag to each instruction. Tags are used to identify the instructions that produce the operand values. When an instruction is dispatched from the dispatch queue to one of the reservation stations, a tag is assigned to the dispatched instruction. In the FM9801, a reorder buffer entry is also allocated for an instruction when it is dispatched. We use the index to the allocated reorder buffer entry as the tag of the instruction.

The register reference table, the reservation stations, the CDB, and the reorder buffer cooperate to implement Tomasulo's algorithm. In the following Subsections 5.3.3, 5.3.4, and 5.3.6, we describe the behavior of each component and discuss how Tomasulo's algorithm works.

5.3.3 Register Reference Table

The register reference table keeps track of instructions that will modify registers. When an instruction is dispatched, the tag of the dispatched instruction is stored in the register reference table entry associated with its destination register. By doing this, the register reference table keeps track of the instructions that will produce the newest register values.

The general-purpose register file and the special register file have separate

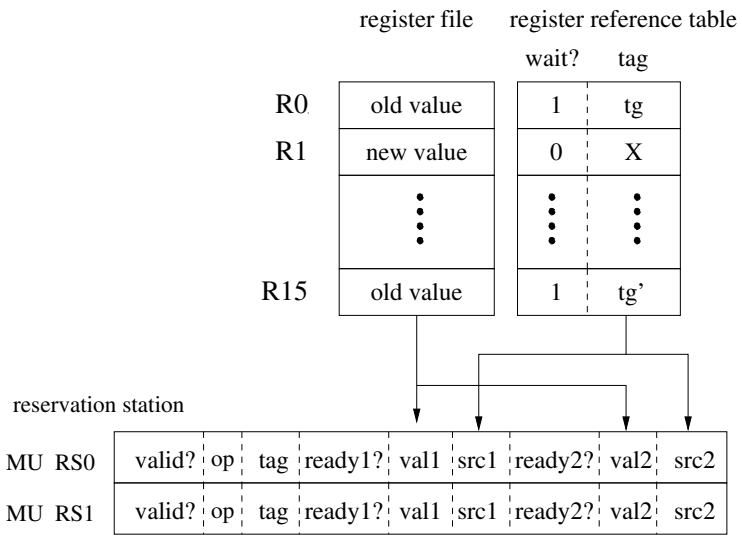

Figure 5.5: The register file, the register reference table and a reservation station.

register reference tables. Figure 5.5 shows the general-purpose register file and its register reference table, as well as the reservation station attached to the multiply unit which we will discuss in the following section. Each register has a corresponding register reference table entry with fields *wait?* and *tag*. The field *wait?* is set to 1 when there are dispatched instructions that will produce a new register value. The field *tag* contains the tag of the most recently dispatched instruction that will update the register. As mentioned earlier, the index to the reorder buffer entry assigned to the instruction is used as its tag in the FM9801. For example, in Fig. 5.5, we assume that some dispatched instruction will update register 0, but no instruction modifies register 1. The *wait?* fields record which registers have a pending write. The instruction that will modify register 0 has tag *tg*.

The register reference table should be updated as follows. When the processor dispatches instruction i that will modify register r, the *wait?* field for register r is set to 1, and the *tag* field is overwritten with the tag of instruction i. Since the instructions are dispatched in order, simply overwriting the *tag* field with the

tag of the newly dispatched instruction will keep track of the instruction that will produce the newest register value. When the register is updated with the new value produced by this instruction, the *wait?* field is reset to 0.

5.3.4 Reservation Station and Common Data Bus

The reservation station is where instructions wait for their operands. The key technique is associating each operand with the tag of the instruction that will produce the operand value. All reservation stations in the FM9801 work on the same principle. As an example, we closely describe the reservation station attached to the multiply unit.

Figure 5.5 shows the reservation station attached to the multiply unit. The reservation station has two entries, RS0 and RS1. The field *tag* of an entry records the tag of the instruction which is stored in the entry. The fields *ready1?* and *ready2?* indicate the availability of the operand register values specified by the instruction fields *ra* and *rb*, respectively. If the flags are 1, the fields *val1* and *val2* store the values of the corresponding operand registers. If they are not, the fields *src1* and *src2* hold the tags of the instructions that will produce the operand values.

When an instruction is dispatched, the *wait?* fields of the register reference table are looked up to determine whether there are pending writes to the operand registers. If no writes are pending, the register values are sent to the fields *val1* and *val2* in the reservation station. Otherwise, the *tag* fields of the register reference table record the tags of the instructions that will produce the operand value. These tags are sent to the *src1* and *src2* fields of the reservation station.

The instruction in the reservation station waits for its operand values to be generated by previous instructions and forwarded through the CDB. Figure 5.6 shows the CDB and the data forwarding logic. The CDB has four buses, *CDB-ready?*, *CDB-tag*, *CDB-val*, and *CDB-excpt*. The bus *CDB-excpt* is not shown in the figure,

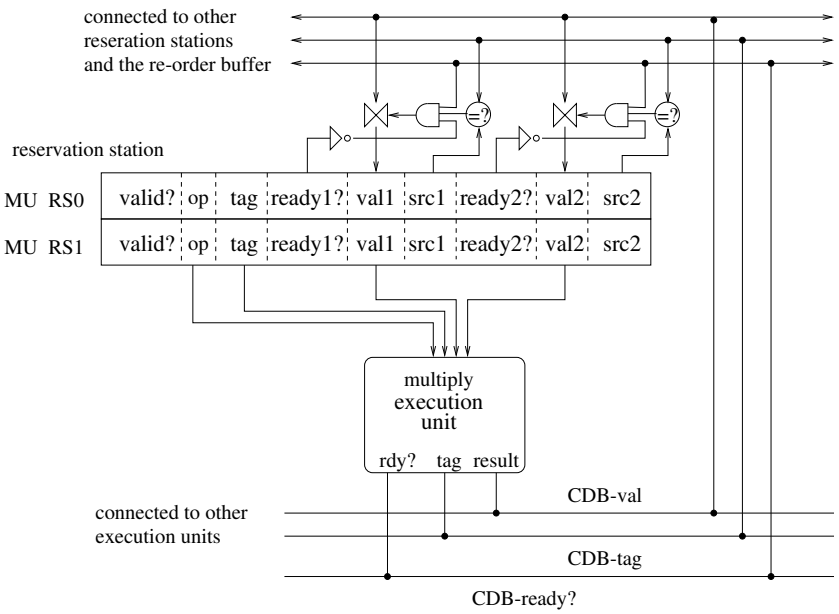

Figure 5.6: A reservation station, the multiply unit and the CDB.

because it is not used for data-forwarding. When the execution of an instruction is completed, the execution unit sets the buses *CDB-ready?* to 1, *CDB-val* to the result, *CDB-tag* to the tag of the completing instruction, and *CDB-excpt* to its exception status.

The reservation station compares the tag in its *src1* and *src2* fields with the tag on the bus *CDB-tag*. When these tags match, the reservation station reads the result on the bus *CDB-val* into the appropriate field *val1* or *val2*. A tag match indicates that the completing instruction is the one that produces the operand value.

When both operands become ready, the instruction is issued to the execution unit. Since the instruction is issued as soon as operands become ready, instructions can be issued out of order.

5.3.5 Execution Units

Instructions are processed in the execution unit appropriate for the instruction type. The integer unit executes ADD, MFSR, and MTSR instructions. The branch instruction handles BR instructions and determines whether conditional branches are taken. The multiply unit processes the MUL instruction with a three-stage pipelined multiplier. The load-store unit executes memory-access instructions such as LD, ST, LDI, and STI. We discuss the load-store unit in Subsection 5.3.9. SYNC and RFEH instructions have no corresponding execution unit.

5.3.6 Reorder Buffer

The reorder buffer is the place where instructions wait to be committed. The execution units may complete the execution of instructions out-of-order, but they are committed in-order after the reorder buffer recovers the original order of the instructions. The reorder buffer is also used for the following purposes:

- Implementing register renaming, and
- Implementing precise exceptions and interrupts.

Register renaming is a technique to permit additional registers to be associated with the architected registers dynamically. A reorder buffer implements register renaming by providing additional space to store register values temporarily. A reorder buffer also helps to implement precise exceptions by forcing instructions to commit in order. We will discuss precise exceptions in Subsection 5.3.8.

The reorder buffer in the FM9801 is a circular buffer with 8 entries. When an instruction is dispatched, the reorder buffer entry is allocated for the dispatched instruction. The FM9801 uses the index to the allocated reorder buffer entry as the tag of the instruction for Tomasulo's algorithm. When an execution unit completes

the execution of an instruction, the result of the execution is temporarily stored in the allocated reorder buffer entry.

The instructions are committed from the head of the reorder buffer. The result of the instruction is copied from the reorder buffer to the destination register when the instruction is committed. Since instructions are committed in order, the register file always appears as if instructions were executed sequentially.

5.3.7 Speculative Execution

The FM9801 implements *speculative execution* with a *branch predictor*. Speculative execution allows the processor to execute instructions beyond a conditional branch before it is determined whether the branch is taken. The branch predictor decides whether the branch is likely to be taken. The processor speculatively executes instructions from the branch target which is more likely to be taken. Branch prediction is performed when a branch instruction is at the IFU unit.

In the FM9801 project, we did not verify the correctness of a branch predictor implementation. Our branch predictor is modeled to nondeterministically return 1 or 0, which indicate that the branch is likely to be taken or not taken, respectively. This suits our verification purposes because we want to verify that the FM9801 correctly executes instructions regardless of the accuracy of the branch prediction.

The branch predictor only predicts whether a branch is likely to be taken. It is the branch execution unit that determines whether a branch is actually taken. If the branch prediction turns out to be incorrect, speculatively executed instructions are abandoned when the branch instruction is committed, and the processor resumes the execution from the correct branch target. If another branch instruction is fetched during the speculative execution, the FM9801 further predicts whether this branch is likely to be taken and continues the speculative execution.

5.3.8 Implementation of Exceptions and Interrupts

The FM9801 implements *precise exceptions* and *precise interrupts* [SP85]. Precise exceptions and interrupts allow the processor to resume the program execution from the point where it is interrupted. Precise exceptions and interrupts should satisfy the following conditions[2]:

- All instructions preceding an exception or an interrupt should be completely executed before the exception handling starts.
- All partially-executed subsequent instructions should be abandoned without any side-effects on the programmer visible states.
- The interrupted instruction may or may not have been executed depending on the type of the exception. Its execution should be completed or totally abandoned.

The FM9801 implements precise exceptions using the reorder buffer. When an instruction is committed in the reorder buffer, the processor checks whether the instruction has raised an exception. If it has, the FM9801 abandons all the subsequent instructions and starts the execution of an exception handler. Since the reorder buffer commits instructions in order, the register file always appears as if instructions were executed sequentially. This enables the FM9801 to satisfy the conditions for precise exceptions without restoring an old state of the register file.

A mispredicted branch and an exception are handled in similar ways. Both require abandoning subsequent instructions. A similar action is needed to handle the instructions that synchronize the machine. In the FM9801, the SYNC and RFEH instructions flush and synchronize the pipeline. We call them *context synchronizing* instructions. Later during the verification, we generalize the concept of speculative

[2] An exception that does not satisfy these conditions is *imprecise*. The behavior of an imprecise exception cannot be rigorously specified. It is not our interest to verify such an ambiguous behavior, and we will consider only precise exceptions in this dissertation.

execution which is caused not only by a mispredicted branch, but also an exception or a context synchronizing instruction.

An external interrupt is handled in a slightly different way. An external interrupt is asynchronous; a microprocessor can interrupt any instruction in the pipeline as long as the interrupt is processed in a reasonably prompt manner. In fact, the FM9801 can interrupt the instruction in the middle of the pipeline. Upon receiving an interrupt signal, the following steps are taken to interrupt an instruction:

1. Halt further instruction dispatching to the reservation stations.
2. Wait until all dispatched instructions retire.
3. Interrupt the oldest undispatched instruction in the pipeline. If no instructions are in the pipeline, the next instruction to be fetched will be interrupted.

This process ensures that the two condition for the precise exception are satisfied.

5.3.9 Memory Access by the FM9801

The load-store unit in the FM9801 handles memory access instructions. The basic organization in the load-store unit is shown in Fig. 5.7. It has a two-entry write buffer, a single entry read-buffer, a result latch, and a two-entry reservation station.

When a load instruction, either LD or LDI, is issued from a reservation station, it advances to the read buffer. Since the memory takes an indefinite number of clock cycles to respond, the load instruction waits for the memory to return the value in the read buffer. The memory protection is checked at this time. When the memory value is returned, the loaded value is sent to the result latch. Finally, the result latch places the loaded value on the CDB, and the execution of the load instruction completes.

A store instruction, either ST or STI, takes the following steps in the load-store unit.

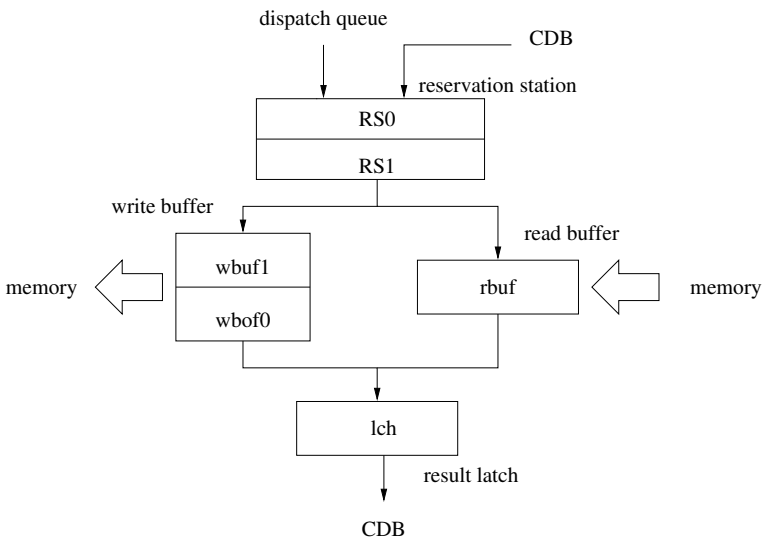

Figure 5.7: The load-store unit of the FM9801.

1. When issued from the reservation station, a store instruction advances to the write buffer. The write buffer is a FIFO queue. The entry *wbuf0* always holds the value for an older store instruction than the entry *wbuf1*.

2. Memory protection is checked, and a data-access error exception is detected if it is write-protected.

3. The exception status is sent to the result latch. The write-buffer continues to hold the address and data which will be used for the memory write operation.

4. The exception flags for the instruction are sent through the CDB to the reorder buffer.

5. The store instruction is committed while the associated memory-write operation is still buffered in the write-buffer.

6. The memory-write operation in the write buffer is actually performed on the memory, and the related information is removed from the write buffer. We say the store instruction is *released* from the write buffer.

This complex process for a store instruction is needed to implement precise exceptions. Since the modification of the memory is not easy to undo, a memory-write operation should not be performed until the processor determines that no preceding exceptions have occurred. We postpone a memory-write operation until after the corresponding store instruction is committed, at which time the instruction is known to be executed safely.

The load-store unit implements a couple of optimizing techniques to improve the performance of the processor. When load and store instructions are executed simultaneously, memory access may be performed out of the original program order, and priorities are given to the load instructions. Because the load instructions may supply operands to the subsequent instructions, this is likely to improve the performance of the processor. The technique is called *load-bypassing*. Load-bypassing is performed only when the memory access address of the load and store instructions are different.

If store and load instructions access the same memory address, and if the store instruction is followed by the load instruction, the stored value is the result of the load instruction. The load instruction can "steal" the operand value of the store instruction, so that the load instruction can be executed without accessing the memory at all. The FM9801 implements this optimization technique called *load-forwarding*.

5.3.10 Formal Specification of the FM9801 Microarchitecture

We formally define the FM9801 microarchitecture using the ACL2 logic, as we have done for the ISA in the last section. We can use the ACL2 definition of the MA to simulate the design, and by simulation we eliminated most of the design faults before attempting its formal verification. Being able to use the same MA definition for simulation and verification is a major advantage of having an executable formal

specification.

The next MA state function, MA-step(*MA*, *sigs*), takes the current MA state, *MA*, and the input signals, *sigs*, and returns the MA state after one clock cycle. Input signals, *sigs*, are represented with the structure type *MA-input*, which is defined in Appendix B.4. The structure type *MA-input* includes the following signals:

- External interrupt signal,
- Acknowledgment from the memory system for an instruction fetch,
- Acknowledgment from the memory system for a data access, and
- An oracle to determine the current branch prediction.

An external interrupt signal initiates interrupt handling. The acknowledgments from the memory are used to model the asynchronous behavior of the memory. When an instruction or a datum is returned from the memory to the processor, the acknowledgment is set to 1. An oracle is used to model the nondeterministic results from the branch predictor. Our branch predictor model returns the provided oracle as the "prediction" result. By verifying that the machine correctly operates for all possible oracle sequences, the machine is guaranteed to work for any branch predictor.

```
DEFINITION:
MA-step (MA, sigs)
def
=
MA-state (step-pc (MA, sigs),
          step-RF (MA),
          step-SRF (MA, sigs),
          step-IFU (MA, sigs),
          step-DQ (MA, sigs),
          step-ROB (MA, sigs),
          step-IU (MA, sigs),
          step-MU (MA, sigs),
          step-BU (MA, sigs),
          step-LSU (MA, sigs),
          step-mem (MA, sigs))
```

The definition of MA-step(MA, *sigs*) is given above. The next state of the MA machine is defined by specifying the next states of individual components, such as the program counter, the register file, and so on. The next component state functions are defined by further specifying the next states of subcomponents. For example, the next-state function, step-LSU (MA, *sigs*), is defined by specifying the next states of the subcomponents of the load-store unit, such as the reservation station, the write buffer, the read buffer, and the result latch.

```
DEFINITION:
step-LSU (MA, sigs)
def
=
let LSU be MA.LSU
    in
load-store-unit (step-RS1-head? (LSU, MA, sigs),
                 step-LSU-RS0 (LSU, MA, sigs),
                 step-LSU-RS1 (LSU, MA, sigs),
                 step-rbuf (LSU, MA, sigs),
                 step-wbuf0 (LSU, MA, sigs),
                 step-wbuf1 (LSU, MA, sigs),
                 step-LSU-lch (LSU, MA, sigs))
```

The function MA-stepn(MA, *sig-list*, n) defines the MA state after n steps of MA execution. The argument *sig-list* is the list of signals to the MA model for each step.

```
DEFINITION:
MA-stepn (MA, sigs-lst, n)
def
=
if n ≃ 0 then MA
 else MA-stepn (MA-step (MA, car (sigs-lst)), cdr (sigs-lst), n − 1)
fi
```

The next ISA state function and the next MA state function are specified in significantly different styles. The next ISA state function focuses on specifying the effects of individual instructions, while the next MA state function is defined by specifying the behavior of each microarchitectural component. There exists a complex time abstraction between the ISA and the MA machines, which complicates

the verification problem. In the following chapters, we consider how to relate the states of these two different models, and how to verify this relationship.

We conclude this chapter by presenting two more definitions needed for the verification of our MA design. The function proj(MA) projects an MA state to an ISA state by removing the states of components invisible to programmers. The predicate flushed-p(MA) is true when the state MA is a flushed state, that is, no partially executed instructions are in the pipeline of the MA state. In the following definition, flushed-p(MA) checks all components are empty and no external interrupt is pending. The function bs-and returns 1 iff all its arguments are 1.

DEFINITION:
proj (MA) $\stackrel{def}{=}$ ISA-state(MA.pc, MA.RF, MA.SRF, MA.mem)

DEFINITION:
MA-flushed? (MA)
$\stackrel{def}{=}$
bs-and (IFU-empty? (MA.IFU),
DQ-empty? (MA.DQ),
ROB-empty? (MA.ROB),
ROB-entries-empty? (MA.ROB),
IU-empty? (MA.IU),
MU-empty? (MA.MU),
BU-empty? (MA.BU),
LSU-empty? (MA.LSU),
b-not (exintr-flag? (MA)))

DEFINITION:
flushed-p (MA) $\stackrel{def}{=}$ MA-flushed? (MA) = 1

Chapter 6

Correctness Criteria for Pipelined Machines

This chapter discusses correctness criteria for microprocessor verifications. Our goal is to verify pipelined machines with various features. We first look at the correctness of machines that execute instructions sequentially, and then discuss the correctness criteria for pipelined machines. We also discuss what should be considered when features such as speculative execution and exceptions are added to pipelined machines.

6.1 Commutative Diagram

A widely used goal of microprocessor verification is to show that the implementation machine behaves as defined by its specification. Particularly, we are interested in showing the execution results of instructions in the implementation machine are exactly the results defined by the specification. This is our primary goal of verification.

In this dissertation, our specification machine is the ISA. The ISA specifica-

tion defines the behavior of the microprocessor from the programmer's viewpoint, and it contains only the programmer visible components. It executes exactly one instruction at every state transition. In this way, the ISA specification defines the effect of individual instructions. Our implementation machine is the MA design. The MA is the clock-cycle accurate model of the actual hardware implementation. It specifies the behavior of microarchitectural components in every clock cycle as discussed in the last chapter.

A *commutative diagram* is often used to represent the correspondence between the MA design and its ISA specification. If the MA executes one instruction every machine cycle, the simple diagram in Fig. 6.1 can express the correspondence of the two machines. In this figure, the solid arrow represents a state transition at the corresponding machine level. The dashed arrow represents the state projection from an MA state to an ISA state; we remove from the MA state the invisible component states which are not part of the ISA. The commutative diagram in Fig. 6.1 compares two paths from the initial MA state MA_0 to the final ISA state ISA_1; one path runs the MA for one machine cycle to get to MA_1, which is then projected to the ISA state, while the other path first projects MA_0 to ISA_0 and steps the ISA once. Since the ISA and the MA execute the same instruction, both paths should result in the same state ISA_1, provided that the MA correctly implements the ISA. [1]

If an MA design takes more than one clock cycle to execute a single instruction, we need to use a slightly more complex diagram. We need to compare multiple steps of MA execution with a single step of ISA execution. This is depicted in Fig. 6.2. Since the machine still executes the same single instruction at both levels, it must obtain the same results. For example, the verification of the FM8501 used this diagram[Hun94].

[1] This diagram may not hold if MA_0 is an unreachable illegal state.

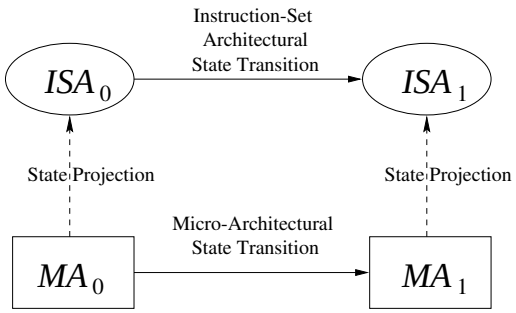

Figure 6.1: Commutative Diagram without Time Abstraction

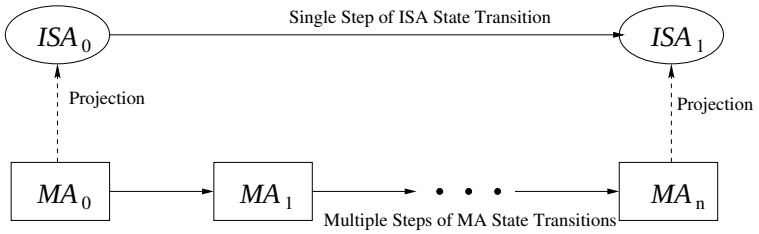

Figure 6.2: Correctness diagram for a machine that takes n cycles to execute a single instruction.

The MA model and the ISA model take different number of machine cycles to execute a single instruction. The ISA model simplifies the behavior of the MA, which may take a varying number of machine cycles to execute a single instruction. This elimination of the timing detail is called *timing abstraction*. *Data abstraction* removes the detail of data representation. For instance, using rational numbers to represent IEEE floating point numbers instead of binary representations is data abstraction. *Control abstraction* hides the detail of control logic. Between our ISA and MA models, there exist timing abstraction and control abstraction, but not data abstraction. Bridging the timing abstraction between the ISA and the MA is a critical problem in pipelined machines, as we will discuss in the next section.

Commutative diagrams can be extended in vertical directions. For instance, the verification of the FM9001 uses four levels of hardware specification: specification level, two-valued logic level, four-valued logic level, and the net-list level. The

specification and two-valued logic levels correspond to our ISA and MA, respectively. By showing that each level is equivalent to the adjacent level, Hunt and Brock have shown that the actual hardware implements their specification [HB92]. On top of the verified microprocessor design, an assembler and a compiler are proven to be correct[Coh86, Moo96]. These multiple layers of verified hardware and software is called the CLI stack.

6.2 Earlier Approaches for Pipelined Machines

Simple commutative diagrams presented in Fig 6.1 and 6.2 do not apply to pipelined implementations. Unlike sequential execution, pipelined machines start executing instructions before the completion of previous instructions. Typically, an MA state contains partially executed instructions in the pipeline. As a result, a simple projection of an MA state may not correspond to any ISA state observed during the ISA execution.

In order to illustrate the problem concretely, we introduce a simple three-stage pipelined machine. Figure 6.3 shows a block diagram of the machine. This machine has three stages: fetch, execution, and write-back. These stages are separated by pipeline latches named "latch1" and "latch2". The fetch stage fetches an instruction and reads operands from the register file (RF). The program counter is updated at this time. The execution stage executes the instruction, and performs the data-access to the memory. The memory modification can take place only in the execution stage. The write-back stage stores the result of the instruction into the register file.

Let us consider the execution of instructions i_0, i_1, i_2 and i_3, in that order. A *reservation table*[Cra96] in Table 6.1 shows how instructions advance through the three-stage pipeline as the time progresses. For instance, instruction i_0 is fetched and stored in latch1 at time t_1, it advances to the latch2 at time t_2 and completes

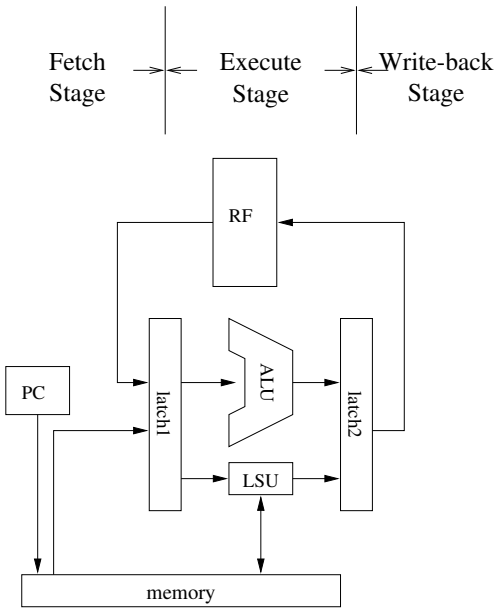

Figure 6.3: An example pipelined machine .

Time →	t_0	t_1	t_2	t_3	t_4
i_0		latch1	latch2		
i_1			latch1	latch2	
i_2				latch1	latch2
i_3					latch1

Table 6.1: Reservation table for the simple pipelined machine.

its operation by time t_3.

In this example execution, the effect of instructions appears on programmer visible components with *timing delays*. For instance, the program counter is updated between time t_0 and t_1 due to the instruction fetch of i_0. Instruction i_0 can only modify the memory between time t_1 and t_2, and the register file between t_2 and t_3. Because of the timing delays between microarchitectural components, states of different components in the MA are related to different ISA states.

This timing delay is best expressed with Fig. 6.4. The MA state at time t is

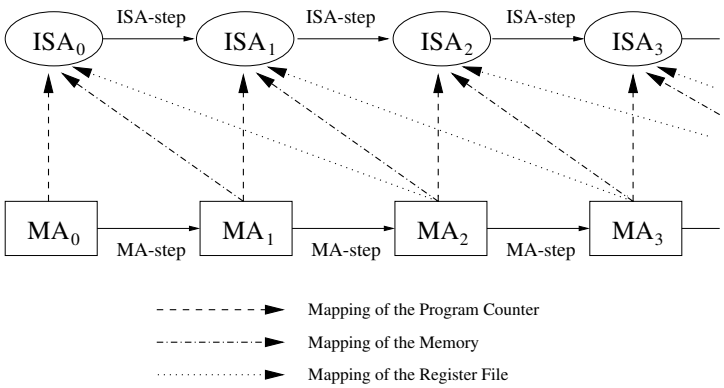

Figure 6.4: Relation of ISA and MA states with timing delay.

represented as MA_t in the figure. The program counter value in ISA_t is the same as that in MA_t, but the memory state in ISA_t is the same as in the next state MA_{t+1}, and the register file state is the same as in MA_{t+2}.

One approach to relate ISA and MA states is to use a *skewed abstraction function*[SM95]. Instead of directly mapping an MA state to an ISA state, a skewed abstraction function relates the states of individual microarchitectural components at different time to a single ISA state. For the example in Fig. 6.4, the abstraction function maps the current program counter value in the MA at time t, the register file at time $t+2$ and the memory at time t to the ISA state at time t. The relation between the MA and the ISA can be checked by verifying the following equality for every t:

$$ISA_t = \text{ISA-state}(MA_t.\text{pc}, MA_{t+2}.\text{RF}, MA_{t+1}.\text{mem}),$$

where function ISA-state constructs an ISA state from individual component states.

Even though the skewed abstraction function technique works with a simple pipelined machine, it is difficult to define such an abstraction function for a complex pipelined architecture. The skewed abstraction function needs to specify the timing delays explicitly. Timing delays vary because of pipeline stalls and non-predictable external interactions such as memory accesses. Since the skewed abstraction func-

tion should take into account all factors affecting the delays, its definition becomes complex and large for a realistic pipelined machine. It is also vulnerable to minor design changes in the pipelined machine implementation. All of the pipelined machines verified with the skewed function technique have a short pipeline with few stalls[TK94, Coe94, WC95, SM95].

Burch and Dill introduced the *pipeline flushing diagram* [BD94] for expressing the correctness of pipelined machines. Fig. 6.5 shows a pipeline flushing diagram. Pipeline flushing is performed by executing the pipelined MA without fetching new instructions until all the partially executed instructions complete. This diagram contains two paths from MA_0 to ISA_n; Burch and Dill called the lower and upper paths *implementation-side path* and *specification-side path*, respectively. On the implementation-side path, we run the MA for one machine cycle from MA_0 to MA_1, flush the pipeline to obtain MA_1', and project it to ISA_n. The specification-side path flushes the pipeline to get to MA_0', projects it to ISA_0, and runs the ISA machine for k cycles, where k is the number of instructions fetched during the transition from MA_0 to MA_1. Typically, k is 1 or 0, depending on whether an instruction is fetched or not during the transition from MA_0 to MA_1. Since both paths completely execute all the partially executed instructions in initial state MA_0 plus k more instructions, the resulting state ISA_n obtained by following two different paths should be the same if the MA design implements the ISA correctly. Let flush(MA) denote the machine state obtained by pipeline flushing from state MA. Using the state transitions functions for the MA and the ISA models introduced in the last chapter, the flushing diagram can be expressed as:

$$\text{proj}(\text{flush}(\text{MA-step}(MA, \mathit{sigs}))) = \text{ISA-stepn}(\text{proj}(\text{flush}(MA)), \mathit{intr\text{-}list}, k).$$

Since we have not yet considered external interrupt here, we assume that the external input signal *sigs* and *intr-list* do not cause external interrupts. The equation should hold even for pipelined machines that execute instructions out of program order,

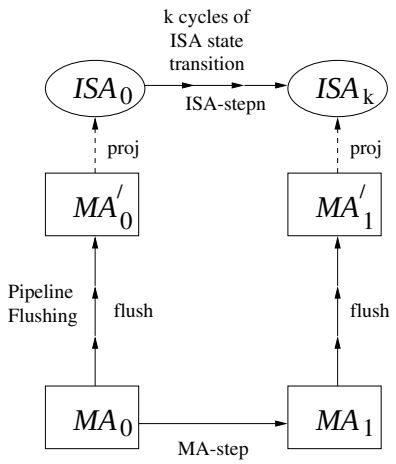

Figure 6.5: Burch and Dill's Flushing Diagram.

because the result of out-of-order execution should appear as if the instructions were executed sequentially. The equation also holds for superscalar machines, for which k can be larger than 1 [Bur96, WB96].

The advantage of Burch and Dill's approach is the use of the pipelined implementation itself as an abstraction function from an MA state to an ISA state. This simplifies the construction of the relationship between the ISA specification and the MA design. Their approach also employs a symbolic simulation technique using so called *uninterpreted functions*. Uninterpreted functions are functions whose definitions are not elaborated. By expressing the output of data-path logic with uninterpreted functions, their technique symbolically simulates machine executions, and syntactically compares the expressions representing the resulting states[2].

In Burch and Dill's approach, the size of the expression representing the simulation result may explode as the number of machine cycles grows, and the pipeline flushing can take many cycles to complete. It also verifies the entire architecture

[2]The idea of uninterpreted functions had been frequently used in the theorem proving community. In Nqthm [BM88], the user can use command *toggle* to effectively convert a function into an uninterpreted function. In ACL2, the user can *disable* a function definition to obtain the same effect. Uninterpreted functions were used extensively in the verification of the FM8501 [Hun94] and FM9001 microprocessors [HB92].

directly without decomposing the design into pieces. As a result, their approach is intractable for complex pipelined designs. Moreover, their approach must assume that the initial state MA_0 satisfies certain invariant conditions, because the flushing diagram may not hold if MA_0 is an inconsistent machine state. Burch and Dill did not mention how to find or verify these invariant conditions in their original paper. In fact, we believe finding and verifying invariant conditions are the most difficult part in the verification of hardware. There have been a number of studies to improve the efficiency of Burch and Dill's verification approach[JDB95, VB98].

Even though Burch and Dill's flushing diagram can be applied to a wide range of pipelined designs including superscalar processors, there are limitations in its applicability. The flushing diagram cannot be directly applied to processors with speculative execution. In the original flushing diagram in Fig. 6.5, the number k represents the number of instructions fetched during the transition from MA_0 to MA_1. However, this number can be incorrect because a processor may fetch instructions speculatively and abandon them later. One modified definition of k is the number of fetched instructions that will be completed by the end of pipeline flushing. However, processors with a branch prediction mechanism require a more complicated definition of k, because the fetched instructions may or may not be completed depending on the prediction.

The pipeline flushing diagram does not apply to machines with external interrupts, either. The ISA specifies the correct behavior for external interrupts: when the ISA receives an external interrupt signal, it immediately interrupts the next instruction to be executed. In the pipelined MA model, however, the machine takes many cycles to synchronize the pipeline and then interrupts an instruction as discussed in the last chapter. Which instruction is actually interrupted depends on the MA implementation.

In fact, the FM9801 may interrupt an instruction in the middle of the

pipeline. Because of this, the flushing diagram may not hold. For example, let us consider the following scenario. Suppose an instruction j is fetched during the single MA step from MA_0 to MA_1. Further assume that an external interrupt signal is sent to the MA during this step. Because of the way the MA design processes an interrupt signal, this may interrupt an earlier instruction i, instead of j. However, in the corresponding ISA execution from ISA_0 to ISA_1, the only instruction that can be interrupted by an external signal is j. Furthermore, the ISA state obtained by following the implementation side throws away all subsequent instructions of i, which are executed in the specification side. Thus, it is not straightforward to check the correctness of external interrupts with the flushing diagram.

6.3 Correctness Criterion for Pipeline Machines

We want to define a correctness criterion which is applicable to a wide range of pipelined machines. Our targets of verification are pipelined machines that may execute instructions out-of-order and speculatively with internal exceptions and external interrupts. The correctness criteria discussed in the previous section do not apply to these types of pipelined machines. We also want our correctness criterion to be implementation independent. For example, the definition of a skewed abstraction function heavily depends on the hardware design. This makes the correctness criteria complex and vulnerable to minor design changes in the implemented hardware. It is hard to trust a correctness criterion itself if its definition is complex. In the rest of this section, we will present our approach to the correctness criterion. For the moment, we assume our processors do not have external interrupts and self-modifying code. We discuss these issues in the following sections.

Figure 6.6 shows our correctness diagram. The diagram defines the validity of arbitrary MA executions that start and end with flushed pipeline states. There are two paths in the diagram. The lower path runs the MA for n clock cycles from

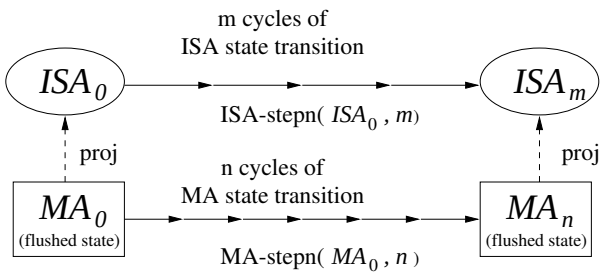

Figure 6.6: Correctness diagram. This diagram compares n cycles of MA execution and m cycles of ISA execution, where m is the number of instructions executed during the MA execution.

a flushed pipeline state MA_0 to another flushed state MA_n. The final flushed state MA_n is then projected to ISA_m. Suppose this n-step MA state transition executes m instructions. The upper path first projects MA_0 to the initial ISA state ISA_0 and then runs the ISA specification for m steps to get the final state ISA_m. If the two paths lead to the same ISA_m for arbitrary n-step MA executions for every n, we say that the MA design always executes instructions as specified by the ISA.

We use the n-step MA function MA-stepn(MA, n), the ISA n-step function ISA-stepn(ISA, n), the projection function proj(MA), and flushed state predicate flushed-p(MA) in the following definition of our correctness criterion. We will not consider external interrupts for the moment, and in this section we use a simplified version of n-step functions which take the current states and numbers of steps, but not external input signals. We assume no instructions are interrupted.

Criterion 1 *(Correctness Criterion without Interrupts) There exists a witness function W_N, and*

$$\text{flushed-p}(MA_0) \wedge \text{flushed-p}(\text{MA-stepn}(MA_0, \textit{sig-list}, n))$$
$$\rightarrow$$
$$\text{proj}(\text{MA-stepn}(MA_0, n)) = \text{ISA-stepn}(\text{proj}(MA_0), W_N(MA_0, n))$$

for any initial state MA_0 and natural number n.

The witness function $W_N(MA_0, n)$ returns the number of instructions completely executed during the n-step MA execution from MA_0. Our correctness diagram restricts the initial and the final states to be flushed, i.e., there are no partially executed instructions in MA_0 and MA-step(MA_0, n). Therefore, we do not have to define a skewed abstraction function that maps unflushed pipeline states to ISA states.

Our correctness criterion is more general than Burch and Dill's pipeline flushing diagram, in the sense that every pipelined machine satisfying the flushing diagram satisfies our correctness criterion. This is shown by the following theorem:

Theorem 1 *Let MA_0 be a flushed MA state. Suppose for every i such that $0 \leq i < n$,*

$$\text{proj}(\text{flush}(\text{MA-step}(MA_i))) = \text{ISA-stepn}(\text{proj}(\text{flush}(MA_i)), k_i) \tag{6.1}$$

where k_i is the number of instructions fetched during the machine cycle from MA_i to MA_{i+1}. Given that MA-stepn(MA_0, n) *is also a flushed state, the following equation holds:*

$$\text{proj}(\text{MA-stepn}(MA_0, n)) = \text{ISA-stepn}(\text{proj}(MA_0), \sum_{i=0}^{n-1} k_i). \tag{6.2}$$

Equation (6.1) represents the flushing diagram, and (6.2) implies our criterion with a witness function that returns $\sum_{i=0}^{n-1} k_i$ as its value. The proof of Theorem 1 can be found in Appendix A.1. Conversely, we can prove the flushing diagram from our correctness criterion with a slight modification. The theorem and its proof are provided in Appendix A.2.

As we discussed in the previous section, Burch and Dill's flushing diagram is applicable to superscalar pipelined machines with out-of-order execution. Theorem 1 suggests that our criterion should be also satisfied by superscalar pipelined machines with out-of-order execution. In fact, our criterion can be applied to a wider range

of pipelined designs. For instance, pipelined machines with speculative execution should satisfy our correctness criterion. After a mispredicted branch, the MA may speculatively execute instructions that should not be executed; however, the MA has to abandon these speculatively executed instructions as if these incorrect instructions have never been executed. This can be checked by comparing the behavior of the MA with the ISA, because the ISA never speculatively executes instructions.

Our correctness criterion guarantees that the result returned by the pipelined MA is always the same as specified by its ISA. This is a safety property: incorrect states are not reached by the MA implementation. It does not, however, imply all the properties that the pipelined machine should satisfy. For instance, our correctness criterion does not imply liveness. Our correctness diagram is vacuous if flushed pipelined states are not reachable. In order to show that the verified criterion is not vacuous, we should prove that the processor reaches a flushed state eventually, or at least we should exhibit an instance of execution that reaches a flushed state. These additional properties can be verified separately.

We have seen that the correctness criterion for pipelined MA designs is different from the correctness criterion for MA designs that sequentially execute instructions. Our criterion for pipelined designs compares only the initial and the final MA states with the corresponding ISA states. The intermediate machine states with partially executed instructions are not compared with the ISA states. For MA designs that sequentially execute instructions, the commutative diagram in Fig. 6.2 compares the MA and the ISA states every time an instruction is completed.

Because our correctness criterion does not check intermediate machine states, there is a question whether our correctness criterion assures the same level of confidence for the verified pipelined design as the commutative diagram does for the design that executes instructions sequentially. Our correctness criterion is based on the observation that commercial RISC processors do not guarantee correct I/O

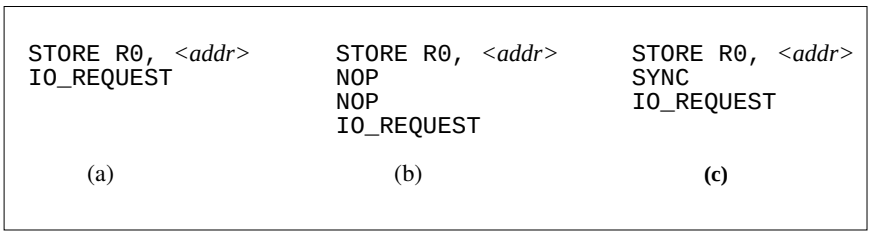

Figure 6.7: Solutions for memory access serialization.

operations unless explicit synchronization is performed. In the rest of this section, we will show how the memory synchronization is performed in such pipelined processors.

Let us consider an example pseudo code shown in Fig. 6.7. Code (a) stores the value of register R0 in the memory at address $<addr>$, and requests an I/O operation. A pipelined machine may not execute this code correctly, because the I/O request can be issued before the STORE instruction completes. As a result, I/O operation may be performed with an incorrect memory value. Code (b) shows an alternative approach for the problem. It issues a few NOP instructions which do nothing but consume clock cycles between the STORE instruction and the I/O operation. We can reduce the chance of incorrect behavior by inserting more NOP instructions. However, this solution depends on the implementation of the pipelined machine. This approach may not work correctly with pipelined machines that execute instructions out of program order, or machines that do not consume many clock cycles for NOP instructions.

The code (c) shows another approach. We introduce instruction SYNC that flushes the pipelined machine. This ensures that all the instructions that precede SYNC will complete their execution before the execution of the subsequent instructions. In this code, the I/O request will not be issued until the STORE instruction writes its value into the memory, thus the I/O operation is correctly performed. Many commercial RISC processors implement instructions with similar effects, such

as the **MB** instruction in Alpha architecture [AAC98] or the **sync** instruction for PowerPC [MSSW94]. They are sometimes called *memory barrier* instructions.

Our correctness criterion compares the states of the MA design with the ISA states only when the pipeline is flushed and synchronized. Our criterion is based on the observation that commercial pipelined processors guarantee the correct values only at the explicit synchronization point. Our correctness criterion assures that we can observe correct machine states through I/O operations only after the machine is explicitly synchronized.

6.4 Exceptions and Correctness Criterion

Exceptions complicate the problem of microprocessor verification. In pipelined machines, multiple instructions may cause exceptions simultaneously, or speculatively executed instructions may cause exceptions that should not occur. Additionally, microprocessors have to provide mechanisms to restart program execution from the point where it is interrupted. This section discusses the verification criterion used for verifying processors with exceptions.

In order to allow the process to be restarted from the point where execution is interrupted, the processor must implement precise exceptions. The two conditions for precise exceptions are given in Subsection 5.3.8. When a pipelined machine encounters an internal exception, the processor flushes the pipeline, stores necessary values into registers, and sets the program counter to the beginning of an exception handler. All preceding instructions are completed, and subsequent instructions are abandoned. This process may take many clock cycles. If multiple exceptions are detected simultaneously, the processor handles only the exception caused by the instruction earliest in program order. A pipelined machine should also ignore a false exception raised by a speculatively executed instruction.

We specified the effects of an internal exception with the ISA next-state

function ISA-step; it takes as argument the ISA state before executing the exception-causing instruction and returns the state right before the execution of the first instruction in the exception handler.

Verifying Criterion 1 can demonstrate that the MA correctly implements precise exceptions for all internal exceptions. Since the ISA model executes instructions one-by-one, it captures the conditions for precise exceptions. Our criterion also implies that the MA correctly handles multiple exceptions that are simultaneously detected in the pipeline because the ISA processes exceptions in program order. We can further check that the MA does not have any side effect from falsely raised exceptions during speculative execution because the ISA never executes instructions speculatively and raises false exceptions. We do not verify whether each exception is correctly processed after the FM9801 vectors to the exception handling routine because this is a software verification issue [SB90].

Let us consider external interrupts. The ISA-step(ISA, $intr$) defined in Chapter 5 takes the current state, ISA, and input signal, $intr$, and returns the next ISA state after executing a single instruction or interrupting an instruction, depending on whether external interrupt signal $intr$ is 0 or 1. On the other hand, MA-step(MA, $sigs$) returns the state after executing the MA for one machine cycle with external signals $sigs$. This MA-step function may not interrupt an instruction immediately even if the external interrupt signal is 1 in $sigs$. It typically takes a number of clock cycles before an instruction is actually interrupted.

Unlike the case for internal exceptions, Criterion 1 does not apply to external interrupts. The diagram in Fig. 6.8 illustrates the execution with external interrupts. The ISA with external interrupts may result in different final states if different instructions are interrupted. For example, the ISA execution starting from ISA_0 may end up with different final ISA states such as ISA_m, ISA'_m, and ISA''_m by interrupting different instructions. On the other hand, the MA design interrupts an

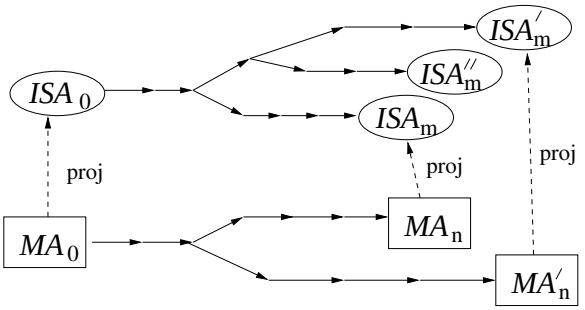

Figure 6.8: Correspondence between the ISA and the MA with exceptions.

instruction in the middle of the pipeline. It is the designer's choice to decide which instruction is interrupted. There is no single correct implementation. It is possible that one MA implementation results in the state corresponding to ISA_m, but another implementation results in another state corresponding to ISA'_m. However, if the ISA and the MA execute and interrupt the same instructions, the resulting states must correspond to each other.

What we prove is that, for an arbitrary MA execution from a flushed state MA_0 to another flushed state MA_n, there is a corresponding ISA execution path such as ISA_0 to ISA_m, and that it satisfies $\mathrm{proj}(MA_0) = ISA_0$ and $\mathrm{proj}(MA_n) = ISA_m$. Trivially, different MA execution paths corresponds to different ISA execution paths. For example, the execution path from MA_0 to MA'_n may correspond to the ISA execution from ISA_0 to ISA'_m. There should be a corresponding ISA execution path for each MA execution path. On the contrary, there may not be any corresponding MA execution path for ISA execution paths such as the execution path from ISA_0 to ISA''_m.

Summarizing the argument so far, we need to prove that, for any MA execution path, there exists a corresponding ISA execution path and our commutative diagram holds. We show the existence of the corresponding ISA execution path for each MA execution path by defining witness functions. Criterion 1 required the existence of a witness function W_N returning the number of instructions executed

by the MA. Here, we use an additional witness function $W_{sig}(MA_0, \textit{sig-list})$ to construct a list of ISA external signals that interrupts the same instructions as the MA does.

Criterion 2 *(Correctness Criterion with External Interrupts) There exist witness functions W_N and W_{sig} such that*

$$\begin{aligned} & (\text{flushed-p}(MA_0) \ \wedge \ \text{flushed-p}(\text{MA-stepn}(MA_0, \textit{sig-list}, n))) \\ \rightarrow \ (& \quad \text{proj}(\text{MA-stepn}(MA_0, \textit{sig-list}, n)) \\ & = \ \text{ISA-stepn}(\text{proj}(MA_0), W_{sig}(MA_0, \textit{sig-list}, n), W_N(MA_0, \textit{sig-list}, n))) \end{aligned}$$

holds for any MA state MA_0, sequence of external signals sig-list, and natural number n.

The witness function W_N here returns the number of instructions executed plus the number of externally interrupted instructions.

This criterion suggests that the MA handles external interrupts precisely as specified by the ISA. However, it does not show other properties about external interrupts. For instance, it does not guarantee that external interrupt signals are eventually processed. Actually, if external interrupt signals are raised too often, the FM9801 cannot process them fast enough, and it will drop some of the interrupt requests. The criterion does not suggest how many cycles it takes before an exception handler is started, either. These properties could be verified independently if necessary.

6.5 Self-Modifying Code in Pipelined Machines

Self-modifying code is an interesting issue in the verification of pipelined machines. All processors, in some sense, must permit the execution of self-modifying programs; just the act of loading a program is one such modification of the program memory.

During the execution of a program by the MA, there are a number of instruction in the pipeline. We call this set of instructions an *execution cloud*. Consider instruction i_0 that modifies the instruction word of i_1. The ISA executes instructions one at a time; i_0 immediately modifies i_1, and when i_1 is executed next time, the processor reads the modified instruction word for i_1. On the other hand, the MA may execute i_0 and i_1 in the same execution cloud. The MA executes i_0 over a number of clock cycles, and i_1 may be fetched before the completion of i_0. As a result, the processor may fetch an unmodified instruction word, and the pipelined MA may execute self-modifying program differently from the ISA.

There are a couple of solutions for the problem of self-modifying code. One approach is dividing the memory into two parts: the program memory and the data memory. This completely eliminates the possibility of self-modifying code. Since this model cannot load a program and execute it, it is not an accurate model of a real processor. However, this approach is a practical compromise between the reality and the abstracted model. This approach has been used to verify several processor models[BD94, SH97].

Another approach to the problem is making the hardware be responsible for detecting self-modifying code. If a self-modifying code is detected, the hardware must stall the execution of the program, so that modifying and modified instructions are not executed in the same execution cloud. Intel's Pentium Processor provides such a hardware mechanism[AA93, Min97]. For this type of processors, we can use the same correctness criterion discussed in the previous sections.

Typical RISC pipelined microprocessors do not have such a detection mechanism of self-modifying code. They require the programmer to explicitly serialize the program execution. Such processors implements instructions to synchronize the program execution similar to the one discussed in Section 6.3.

For instance, we can safely load a program by first loading the code into the

memory, executing the SYNC instruction that causes the pipeline to flush, and then jumping to the loaded program. We assume that no instruction modifies another instruction in a program segment between two consecutive flushed states, thus each segment of the program is executed correctly. We can append multiple segments of such MA execution, and preserve the execution correctness. What the hardware must guarantee is the result of each program fragment is correct as long as no self-modification occurs in the segment

We modify Criterion 2 to cover pipelined machines that may execute self-modifying code without a hardware detection mechanism. First we introduce the predicate ISA-self-modify-p(ISA_0, *intr-list*, n), which checks whether the ISA execution corresponding to ISA-stepn(ISA_0, *intr-list*, n) runs a self-modifying program. With this predicate, our correctness criterion is defined as follows:

Criterion 3 *(Correctness Criterion with Possible Self-modifying Code) There exist witness functions W_N and W_{sig}, and*

$$
\begin{aligned}
&(\quad \text{flushed-p}(MA_0) \\
&\quad \wedge \text{flushed-p}(\text{MA-stepn}(MA_0, \textit{sig-list}, n)) \\
&\quad \wedge \neg \text{ISA-self-modify-p}(\text{proj}(MA_0), W_{sig}(MA_0, \textit{sig-list}, n), W_N(MA_0, \textit{sig-list}, n)))) \\
\rightarrow\ &(\quad \text{proj}(\text{MA-stepn}(MA_0, \textit{sig-list}, n)) \\
&\quad = \text{ISA-stepn}(\text{proj}(MA_0), W_{sig}(MA_0, \textit{sig-list}, n), W_N(MA_0, \textit{sig-list}, n)))
\end{aligned}
$$

for any MA state MA_0, sequence of external signals sig-list, and natural number n.

Intuitively speaking, we assume that the ISA executes no self-modifying program between the initial and the final ISA states, and show the commutative diagram holds in the same way as Criterion 2. This criterion is what we verify for the FM9801.

In this chapter, we have discussed a variety of correctness criteria. No single correctness criterion implies the complete correctness of a verified processor. For

instance, our correctness criteria do not address the liveness issue. The verified criterion may be vacuous unless we show that there exists an MA execution to reach a flushed final state. They do not address other issues like performance verification, the verification of prompt responses for interrupt signals, and correct behaviors in a multi-processor environment.

However, verifying our correctness criteria guarantees that the result generated by the pipelined MA is always the result specified by the ISA. This would not be the case if the MA design causes pipeline hazards, incorrectly implements speculative execution, or incorrectly processes exceptions. Verifying our correctness criteria can reveal subtle designs faults in the pipelined machines, and the verification requires a profound analysis of the behavior of pipelined architecture. In the following chapters, we will discuss the verification of our criterion for the FM9801.

Chapter 7

Intermediate Abstraction

7.1 Purpose of an Intermediate Abstraction

Our intermediate abstraction, which we call *Microarchitectural Execution Trace Table* or simply *MAETT*, is an auxiliary variable. Technically, it is called a *history variable* which records the past behavior of the MA machine [AL91]. Auxiliary variables are often used to ease the definitions of invariants and refinement mappings. Typical pipelined machine implementations are optimized for performance, and not all the information useful for the verification is recorded in the machine state. For instance, the original instruction word is dropped as soon as the instruction is decoded, even though it would be useful to know the original instruction word for verifying the behavior of the machine. We can record such useful information in an auxiliary variable. We are particularly interested in an auxiliary variable which enables us to directly reason about the instructions, because various properties about pipelined execution can be represented as properties of executed instructions.

A MAETT representation mimics a reservation table of pipelined machines. Figure 7.1 shows an example reservation table for the FM9801. This table shows the stages of instructions i_0, i_1, i_2 and i_3 at each machine state. Stages are shown as

	MA_0	MA_1	MA_2	MA_3	MA_4
i_0		(IFU)	(DQ 0)	(IU RS0)	(complete)
i_1			(IFU)	(DQ 0)	(IU RS1)
i_2				(IFU)	(DQ 0)
i_3					(IFU)

Figure 7.1: A reservation table for the FM9801. The MAETT corresponds to a column of a reservation table.

(IFU), (DQ 0) and (DQ 1). For instance, instruction i_0 is at the stage (IFU) in the machine state MA_1 and it advances to the stage (DQ 0) in state MA_2. The MAETT resembles a column of the reservation station. A MAETT records the stages and other related information of the executed instructions.

In the ACL2 definition of the MAETT, a column of the reservation table is represented with an ACL2 list. This list records instructions in program order, where each instruction is represented using an ACL2 structure named INST.

This list representation enables us to define recursive predicates to specify properties on the executed instructions. We will define and verify various invariant properties that are defined as predicates of a MAETT, and use the MAETT as a stepping-stone in the proof of the correctness criterion.

In the rest of the chapter, we will discuss the MAETT abstraction in detail. We discuss the definition of MAETT states for reachable machine states in Section 7.2. In Section 7.3, we look closely into the representation of instructions in the MAETT. Then, we define and prove basic properties of instructions recorded in the MAETT. In Section 7.4, we discuss the program order of instructions. In Section 7.5, we introduce functions that return the instruction at a particular stage. In Section 7.6, functions are defined to specify instructions that generate operand values.

7.2 Data-Structure and Functions for MAETT

We define the MAETT for all reachable MA states. First, we recursively define that an MA state is *reachable by n-steps* as follows:

1. State MA is reachable by 0-step if it is a flushed state, that is, flushed-p(MA) is true. In this case, MA is an *initial state*.

2. State MA' is reachable by $(n+1)$-steps if there exists a state MA which is reachable by n-steps and $MA' = \text{MA-step}(MA, sigs)$ with some external input signals *sigs*.

We say that MA is *a reachable state* iff MA is reachable by n-steps for some natural number n. It is easy to prove by induction that MA_n is a reachable state iff MA_n can be represented as MA-stepn(MA_0, *sig-list*, n) with some flushed MA state MA_0, a list of external signals *sig-list*, and a natural number n.

Now, we recursively define the MAETT state for a reachable MA state with two functions MT-init(MA) and MT-step(MT, MA, *sigs*) as follows:

1. If MA is an initial state satisfying flushed-p(MA), then MT-init(MA) is the MAETT of MA.

2. If state MA' is reachable by $(n+1)$-steps, then there exists a state MA that is reachable by n-steps and $MA' = \text{MA-step}(MA, sigs)$. Let MT be the MAETT of MA. Then, MT-step(MT, MA, *sigs*) is the MAETT of MA'.

The existence of the MAETT states for all reachable states can be proven by induction.

As we have defined an n-step function for MA models, we can define an n-step function for the MAETT abstraction. Suppose MA_0 is a flushed initial MA state, and $MT_0 = \text{MT-init}(MA_0)$. The following function MT-stepn(MT_0, MA_0, *sig-list*, n) defines the MAETT for the reachable state MA-stepn(MA_0, *sig-list*, n).

```
DEFINITION:
MT-stepn (MT, MA, sig-list, n)
def
=
if n ≃ 0 then MT
 elseif endp (sig-list) then MT
 else MT-stepn (MT-step (MT, MA, car (sig-list)),
                 MA-step (MA, car (sig-list)),
                 cdr (sig-list),
                 n − 1)
fi
```

Figure 7.2 shows the data-structures used to define a MAETT state. A MAETT is represented with structure types defined with the ACL2 macro **defstructure**. The structure INST is used to represent the current status of an individual instruction. We will discuss the details of the instruction representation in the following section. The structure MAETT defines the data type for an entire MAETT state. The *trace* field of the MAETT structure stores the true list of retired and in-flight instructions represented using the INST structure. This list records the instructions in program order. The true list of INST structures, INST-listp, is defined using the ACL2 macro **deflist**.

For instance, three instructions i_0, i_1 and i_2 are fetched and being executed in the MA state MA_3 in Fig. 7.1. Let MT_3 be the MAETT representing the state MA_3. The *trace* field of a MAETT MT_3 stores a list $(i_0\ \ i_1\ \ i_2)$, where i_0, i_1 and i_2 are represented using the INST structures. If we use the dot notation introduced in Chapter 3, MT.trace $= (i_0\ \ i_1\ \ i_2)$.

Additionally, the structure used to define a MAETT records other parameters useful in the definition of pipelined machine properties. The fields *DQ-len* and *WB-len* record the number of instructions stored in the dispatch queue and the write buffer, respectively. The fields *ROB-head*, *ROB-tail*, and *ROB-flg* are the head pointer, the tail pointer, and the wrap-around flag for the reorder buffer. The field *new-ID* is used to store the new ID for the newly fetched instruction. The purpose of the ID is discussed in the next section.

```
Defstructure INST {
    naturalp        ID ;              // Identification Number
    bitp            modified? ;       // Modified by Self-Modifying Code?
    bitp            first-modified? ; // First Modified Instruction
    bitp            specultv? ;       // Speculatively Executed?
    bitp            br-predict? ;     // Branch Prediction Result
    bitp            exintr? ;         // Externally Interrupted
    stage-p         stg ;             // Current Stage
    ROB-index-p     tag ;             // Tag used in Tomasulo's Algorithm
    ISA-state-p     pre-ISA ;         // Pre-ISA state
    ISA-state-p     post-ISA ;        // Post-ISA state
    }

Deflist INST-listp as List of INST-p

Defstructure MAETT {
    ISA-state-p     init-ISA ;        // Initial ISA state
    naturalp        new-ID ;          // ID for Newly Fetched Instruction
    naturalp        DQ-len ;          // # of Instructions in Dispatch Queue
    naturalp        WB-len ;          // # of Instructions in Write Buffer
    bitp            ROB-flg ;         // Circular State Flag of Reorder buffer
    ROB-index-p     ROB-head ;        // Head of Reorder Buffer
    ROB-index-p     ROB-tail ;        // Tail of Reorder Buffer
    INST-listp      trace ;           // List of Executed Instructions
    }
```

Figure 7.2: Definition of MAETT data-type.

Finally, the *init-ISA* field of a MAETT stores the initial ISA state. As mentioned earlier, a reachable MA state is represented as MA-stepn(MA_0, *sig-list*, n) and its MAETT is defined as MT-stepn(MT-init(MA_0), MA_0, *sig-list*, n). The state MA_0 is the initial flushed MA state from which program execution starts. We define the ISA projection of MA_0, proj(MA_0), to be the initial ISA state from which the comparable ISA execution starts, and we store it in the *init-ISA* field of the MAETT.

This completes the description of the fields of the MAETT structure. The MAETT initial function MT-init and next-state function MT-step should maintain correct values in these fields. Keeping track of all fetched and executed instructions is a tricky but an essential part of the MAETT abstraction. While maintaining the list of instructions in program order, we also have to update individual INST structures in the list, so that each INST structure represents the current status of the instruction. In the rest of the section, we discuss how the MAETT defining functions MT-init and MT-step maintain the list of retired and in-flight instructions.

The initial MAETT function MT-init(MA) simply sets the *trace* field of an MAETT to an empty list **nil**.

$$\text{MT-init}(MA_0).\text{trace} = \textbf{nil}$$

This means no instructions have been fetched and executed in the initial state MA_0.

In order to define the MAETT next-state function MT-step, we introduce three functions fetched-INST, exintr-INST, and step-INST, which define the states of individual INST structures in the MAETT. The function fetched-INST defines the INST structure representing a newly fetched instruction. The function exintr-INST defines the INST structure representing an externally interrupted instruction. And the function step-INST defines the updated status of an instruction in the next state. More precisely speaking, if INST structure i represents the status of an instruction in the current state MA, step-INST($i, MT, MA, sigs$) defines the status of the same instruction in the next MA state MA-step($MA, sigs$).

Let us consider the current state MA and its MAETT MT. Let MA' be the next MA state MA-step(MA, *sigs*) and MT' be its MAETT MT-step(MT, MA, *sigs*). Suppose MT.trace $= (i_0 \ \ldots \ i_{n-1})$. In other words, $i_0, \ldots, i_{n-1}$ represent the instructions that have been retired or are in-flight. The MAETT next-state function MT-step(MT, MA, *sigs*) adds and deletes elements in the *trace* field and returns an MT' that satisfies:

$$MT'.\text{trace} = \begin{cases} (i'_0 \ \ldots \ i'_k) & \text{If } i_k \text{ flushes subsequent instructions,} \\ (i'_0 \ \ldots \ i'_{k-1} \ i_{intr}) & \text{if instruction } i_k \text{ is externally interrupted,} \\ (i'_0 \ \ldots \ i'_{n-1} \ i_{fetch}) & \text{if a new instruction } i_{fetch} \text{ is fetched, and} \\ (i'_0 \ \ldots \ i'_{n-1}) & \text{otherwise.} \end{cases}$$

In this definition, i_{fetch} is the INST structure defined with the function fetched-INST and represents a newly fetched instruction. The INST structure i_{intr} is defined with exintr-INST and represents an externally interrupted instruction. The INST structure i'_k represents the same instruction as i_k, but it records the status of the instruction in the next state MA' while i_k represents the status in MA. Formally, $i'_k = \text{step-INST}(i_k, MT, MA, sigs)$.

There are four cases in the above definition of MT'.trace. The first case is when the instruction represented by i_k flushes the subsequent instructions. If an instruction is a mispredicted branch, an exception-raising instruction, or a context-synchronizing instruction, the subsequent instructions should not be completed but must be abandoned. Mimicking this behavior of the processor, the function MT-step(MT, MA, *sigs*) removes the INST elements $i_{k+1}, \ldots, i_n$ representing the subsequent instructions, and updates $i_0, \ldots, i_k$ with $i'_0, \ldots, i'_k$.

In the second case, the instruction represented by i_k is externally interrupted. In this case, the subsequent instructions are similarly abandoned, and i_k is replaced with i_{intr}.

In the third case, no instructions are abandoned and a new instruction is fetched. A new instruction represented as i_{fetch} is added at the end of the list. Since the FM9801 fetches instructions in order, adding i_{fetch} at the end of the list maintains the program order of the instructions.

Finally, if no instructions are fetched nor abandoned, MT-step simply updates all INST structures $i_0, \ldots, i_{n-1}$ with $i'_0, \ldots, i'_{n-1}$. Since the retired instructions are not removed from the list, it keeps all retired instructions as well as in-flight instructions.

As the reader may have realized, the definitions of the MAETT data-structure and the next-state functions are directly related to the MA design; a different MA design will have a different MAETT representation. However, maintaining a list of instructions in program order is one of the key ideas for the definition of the MAETT abstraction. This will allow many properties to be defined as recursive predicates as we will see later.

One problem with the MAETT abstraction is that the complex definitions of the MAETT defining functions, MT-init and MT-step, may not be correct. The definition of these functions becomes complex as the original machine design becomes complicated. An answer to this problem is that the MAETT definition will be proven to satisfy various invariant properties that relate it to the original machine state. In the next chapter, we define and verify these invariant properties on the MAETT. If the MAETT defining functions do not correctly emulate the behavior of the MA design, the verification of invariant properties fails and the bugs in the MAETT defining functions are exposed. We iteratively fixed the definition of the MAETT defining functions until we completed the verification.

In the next section, we will see in more detail the representation of the instructions in the MAETT. This will clarify what we record in the MAETT abstraction of an MA state.

7.3 Representation of Instructions

The representation of instructions is a key issue in the MAETT abstraction. We represent the status of each instruction with the structure INST. This structure stores the information about an instruction that may not be kept by the MA design but may be useful in the analysis of the instruction. In defining the structure, we paid attention to two issues: the conciseness of the representation and its expressibility. A simple representation eases the definition of the MAETT abstraction. At the same time, we must be able to define numerous values and check various conditions of the instructions using the INST representation. We will examine this by discussing each field of the INST structure. The definition of the INST structure was given in Fig. 7.2.

In subsection 7.3.1, we define the pipeline stages for instructions. Formally defining pipeline stages with a fine granularity will allow us to precisely reason about the internal latches. Also the stages of the instructions must be known when defining various invariant conditions. In Subsection 7.3.2, we study the ISA states related to each instruction. In Subsection 7.3.3, we discuss how we record speculatively executed instructions, and in Subsection 7.3.4, we describe how self-modifying instructions are recorded. In Subsection 7.3.5, we cover additional issues concerning our instruction representation. From now on, the INST structure i is simply called "instruction i", as long as its implication is clear.

7.3.1 Stages of Instructions

The field *stg* of the INST structure records the pipeline stage of the represented instruction. Formally, we defined 26 pipeline stages for the FM9801. These pipeline stages specify the pipeline latches in which the intermediate results of instructions are located. For instance, instructions in different reservation station entries are considered to be at different stages. In this sense, our pipeline stages have finer

granularity than those typically used in the discussion of pipelined designs.

In the ACL2 logic, we defined these 26 pipeline stages as an ACL2 list constant. The different stages and the paths between stages are shown in Fig. 7.3. For instance, the instruction fetch stage is defined with the constant `'(IFU)`, and the stage for an instruction at dispatch queue entry 0 is `'(DQ 0)`. Directed edges in the figure indicate possible transitions of an instruction from a stage to another stage. For example, an instruction at the `'(IFU)` stage can move to stages `'(DQ 0)`, `'(DQ 1)`, `'(DQ 2)`, and `'(DQ 3)`. Self-pointing edges corresponding to instruction stalls are not shown here, as a stall can occur in any stage. Every instruction starts at the instruction fetch stage, `'(IFU)`, and reaches the retire stage, `'(retire)`, unless it is abandoned due to speculative execution. For example, an ADD instruction may pass through the following stages:

`'(IFU)` $\rightarrow$ `'(DQ1)` $\rightarrow$ `'(DQ0)` $\rightarrow$ `'(IU RS0)` $\rightarrow$ `'(complete)` $\rightarrow$ `'(retire)`.

We divide the pipeline stages into several major groups by defining predicates of stages. Figure 7.3 shows rounded boxes surrounding stages, indicating they are grouped together. The label on the boxes shows the predicate to test whether a stage belongs to that group. For example, DQ-stg-p($'$`(DQ0)`) is true and DQ-stg-p($'$`(IFU)`) is not.

Particularly, committed-p(i) is true if i is a committed instruction, and dispatched-p(i) is true if i is a dispatched instruction. As discussed in Chapter 5, the FM9801 dispatches and commits instructions in order, and this property is critical for the correct operation of the machine.

7.3.2 ISA States and Interrupt Signals

The field *pre-ISA* of INST structure i stores the ideal ISA state before executing the instruction represented by i. We call it the *pre-ISA state* of i. The field *post-ISA* stores the ideal ISA state after the execution, and we call it the *post-ISA state* of i.

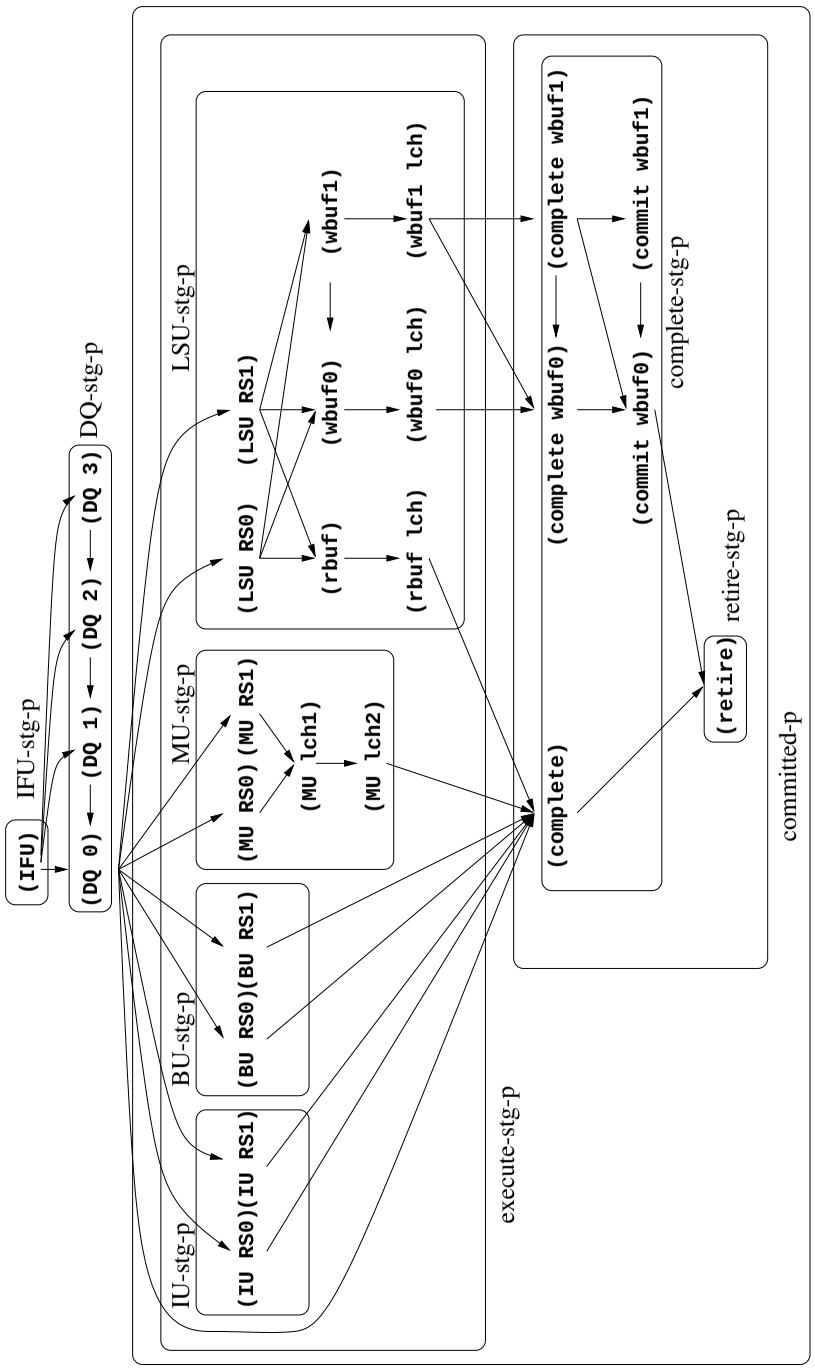

Figure 7.3: Stages of Instructions in the FM9801. Boxes surround the stages satisfying the labeling predicates.

In principle, the pre-ISA state and post-ISA states for a particular instruction can be calculated if we know the initial ISA states and all instructions that have been executed. We also need to know which instructions are interrupted by external signals, because the behavior of the ISA machine changes depending on interrupts.

The field *exintr?* of the INST structure is used to record which instructions are actually interrupted. Like a typical microprocessor design, an external signal of FM9801 may interrupt instructions in the middle of the pipeline, not the instruction that is fetched in the current cycle. As a result, the relation between the timing of an external interrupt signal and the choice of the interrupted instruction is not simple. Our MAETT records which instructions are actually interrupted by setting the *exintr?* field of the INST structure. The MAETT next-state function MT-step sets the field of the interrupted instruction if the instruction is interrupted in the MA execution.

The pre-ISA and post-ISA states of each instruction recorded in a MAETT MT can be defined recursively. Let us represent the k'th instruction recorded in MAETT MT as i_k.

$$i_k.\text{pre-ISA} = \begin{cases} MT.\text{init-ISA} & \text{if } k = 0 \\ i_{k-1}.\text{post-ISA} & \text{if } k \neq 0 \end{cases}$$

$$i_k.\text{post-ISA} = \text{ISA-step}(i_k.\text{pre-ISA}, i_k.\text{exintr?})$$

For example, if $MT.\text{trace} = (i_0\ i_1\ \cdots\ i_{m-1})$, the initial ISA state of the program is recorded as MT.init-ISA. This state is the pre-ISA state of i_0 and we name it ISA_0. The post-ISA state of i_0 is defined with the ISA next-state function as ISA-step(ISA_0, i.exintr?), using the information stored in the exintr? field to determine whether instruction i is interrupted. This post-ISA state is the result of executing instruction i_0, and we name this ISA state ISA_1. It is the state from which the next instruction i_1 is executed. Thus ISA_1 is the pre-ISA state of i_1. Similarly, the post-ISA state, ISA_2, of i_1 can be calculated from ISA_1. In this way, the ISA

state sequence $ISA_0, ISA_1, \ldots, ISA_m$ can be defined. This ISA execution sequence is what the MA is trying to mimic.

The pre-ISA state is particularly useful for defining various values related to the instruction. Following are some of the functions that calculate related values of an instruction from its INST representation i:

$$
\begin{array}{lcl}
\text{INST-pc}(i) & \stackrel{def}{=} & i.\text{pre-ISA.pc} \\
\text{INST-RF}(i) & \stackrel{def}{=} & i.\text{pre-ISA.RF} \\
\text{INST-mem}(i) & \stackrel{def}{=} & i.\text{pre-ISA.mem} \\
\text{INST-word}(i) & \stackrel{def}{=} & \text{read-mem}(\text{INST-pc}(i), \text{INST-mem}(i)) \\
\text{INST-opcode}(i) & \stackrel{def}{=} & \text{opcode}(\text{INST-word}(i)) \\
\text{INST-ra}(i) & \stackrel{def}{=} & \text{ra}(\text{INST-word}(i)) \\
\text{INST-rb}(i) & \stackrel{def}{=} & \text{rb}(\text{INST-word}(i)) \\
\text{INST-rc}(i) & \stackrel{def}{=} & \text{rc}(\text{INST-word}(i)) \\
\text{INST-im}(i) & \stackrel{def}{=} & \text{im}(\text{INST-word}(i))
\end{array}
$$

INST-pc(i) defines the program counter value for i, i.e., the address of the instruction word for i. INST-RF(i) and INST-mem(i) define the ideal register file and the memory state before the execution of instruction i. INST-word(i) defines the instruction word of i, which is the value of the memory addressed by the program counter. INST-opcode(i) defines the opcode of i. INST-ra(i), INST-rb(i), INST-rc(i), and INST-im(i) define the *ra*, *rb*, *rc* and *im* field values in the instruction, as shown in Fig. 5.2.

Similarly we can define more related values for instructions, such as the memory access address or the result of execution. In the FM9801 verification project, we defined 58 such functions, which are listed in Appendix C. Table 7.1 lists some of the functions that will be used later. These functions are particularly useful in defining lemmas and invariant properties for instructions, as we will discuss later.

Function Name	Description
INST-excpt?(i)	1 iff i causes an internal exception.
INST-wrong-branch?(i)	1 iff branch prediction is incorrect.
INST-context-sw?(i)	1 iff i is a context synchronizing instruction.
INST-store?(i)	1 iff i is a memory store instruction.
INST-store-addr(i)	Memory store address of i if i is a store instruction.
INST-dest-reg(i)	Destination register of i.
INST-dest-val(i)	Destination register value for i.
INST-fetch-error?(i)	1 iff fetching i causes a fetch error exception.

Table 7.1: Some functions that calculate values for an instruction represented by i.

7.3.3 Speculatively Executed Instructions

The field *speculv?* of the INST structure is a one-bit flag to indicate whether the represented instruction is being speculatively executed. We say that an instruction is *speculatively executed* iff the MA model fetches and partially executes the instruction but the ISA model does not. For example, instructions following a mispredicted branch are speculatively executed. The ISA model does not fetch these instructions, because branch instructions are completely executed before the next instruction is fetched. However, an MA design with a deep pipeline may fetch and execute many instructions before the branch is known to be taken.

We also consider instructions speculatively executed if they follow an instruction causing an internal exception. The MA of the FM9801 operates under the assumption that an instruction does not raise an exception, and "speculatively" executes the subsequent instructions until an exception is detected. The MA abandons instructions following an exception-raising instruction before they are committed, in a way similar to those following a mispredicted branch. The ISA never executes such instructions because exceptions are processed immediately.

The context-synchronizing instruction discussed in Section 5.3.8 also starts a speculative execution in the same sense as the other two types of instructions. A

context-synchronizing instruction flushes the pipeline and may change the processor's privilege mode. When a context switching instruction commits, the processor abandons all subsequent instructions and restarts execution, in a way similar to mispredicted branch instructions.

By our definition, instructions following a correctly predicted conditional branch instruction are not speculatively executed, because both the ISA and the MA execute them. Instructions following an externally interrupted instruction are not considered to be speculatively executed either, because those instructions would be executed both in the ISA and the MA if the external interrupt did not occur.

We define a function INST-start-specultv?(i) that takes an instruction i in the INST representation and returns 1 if i starts speculative execution. We use a few functions found in Appendix C. Functions INST-excpt?(i), INST-wrong-branch?(i) and INST-context-sw?(i) return 1 when i causes an internal exception, i is a mispredicted branch, and i is a context synchronizing instruction, respectively. [1] Function INST-start-specultv?(i) returns 1 if i falls into one of these instruction types and i is not committed yet. The condition of ¬committed-p(i) is necessary, because the processor does not speculatively execute instructions after the commitment of branch instructions, exceptions, or context switching instructions. For example, the processor will have determined the correct outcome of branch instructions by the time the branch instruction is committed, and it fetches instructions from correct target addresses after the branch is committed.

```
DEFINITION:
INST-start-specultv? (i)
def
=
if committed-p (i)  then 0
 else bs-ior (INST-excpt? (i),
              INST-context-sync? (i),
              INST-wrong-branch? (i))
fi
```

[1] ACL2 macro bs-ior($b_1, b_2, b_3, \ldots$) takes the bit inclusive-OR of all arguments. If any of bits $b_1, b_2, b_3, \ldots$ are 1, it returns 1.

The processor speculatively executes all subsequent instructions of an instruction whose INST representation i satisfies INST-start-specultv?$(i) = 1$. Field *specultv?* of an INST is set to 1 if the represented instruction is speculatively executed. Suppose a MAETT MT records instructions $i_0, \ldots, i_m$; MT.trace $= (i_0 \ \ldots \ i_m)$. Since the list in the *trace* field records instructions in program order, INST-start-specultv?$(i_h) = 1$ implies that i_k.specultv? $= 1$ for all k such that $h < k \leq m$.

7.3.4 Modified Instructions

Self-modification of the program can be a problem in pipelined machine verification. As discussed in Chapter 6, execution of self-modifying code may have different effects on the ISA model and the MA model because a pipelined machine may fetch instructions before the instruction modification is completed. We keep track of which instructions are modified by the program, and verify that unmodified instructions are executed correctly.

We say that an instruction is *directly modified* if its instruction word in the memory is modified by a preceding instruction. We say that all subsequent instructions of a directly modified instruction are *modified*. A modified instruction may be influenced by a preceding directly modified instruction and may not be executed correctly. For example, a self-modifying program may change the course of execution by modifying a branch instruction. In such a case, all subsequent instructions are executed differently from the ISA model.

We can define a predicate INST-modify-p(i, j) which is true iff instruction i modifies another instruction j.

DEFINITION:
INST-modify-p(i, j)
$\stackrel{def}{=}$
  (INST-store?$(i) = 1$)
$\wedge$ (INST-store-addr$(i) =$ ((j.pre-ISA).pc))

$\wedge$ (INST-excpt? $(i) \neq 1$)
$\wedge$ $((i.\text{exintr?}) \neq 1)$
$\wedge$ $((j.\text{exintr?}) \neq 1)$

This predicate checks whether i is a memory-store instruction that writes the memory at the address of the instruction j. It also checks exceptions and external interrupts do not occur because the memory store operation is not executed if an exception is detected or an external interrupt may cancel the execution of a modified instruction.

If a MAETT records a list of instructions $MT.\text{trace} = (i_0\ i_1\ \ldots\ i_m)$ and INST-modify-p(i_j, i_k), then instruction i_j modifies i_k and i_k satisfies i_k.modified? = 1. The MAETT next-state function MT-step scans the list of instructions in the MAETT and properly sets the value of the field *modified?* when a fetched instruction happens to be modified by a previous instruction.

The *first-modified?* field is set to 1 iff the instruction is the first instruction that is modified in the program recorded in the MAETT. Even though it may be executed incorrectly, the first modified instruction holds the correct program counter value, and we need to distinguish it to prove the correctness of program counter values.

7.3.5 Other INST Fields

This section describes the remaining fields that were not discussed in the previous subsections.

Field *br-predict?*: Prediction Result

The *br-predict?* field of INST structure records the branch prediction result for a conditional branch. Since our machine predicts the result of a branch instruction at the `'(IFU)` stage, the field *br-predict?* of INST structure i is set to the output from the branch predictor when i.stg = `'(IFU)`.

Field *tag*: Tag of the instruction

The field *tag* of the INST structure records the tag which is used to identify the instruction for Tomasulo's algorithm. The value in this field is the index to the reorder buffer entry allocated for the instruction, since the FM9801 uses it as the tag. The value stored in this field is used in the verification of Tomasulo's algorithm, which dynamically resolves the data-dependencies between instructions.

Field *ID*: Identity Number

This field is used to store the identifier of the instruction. By assigning distinct ID numbers, we can make the list in the *trace* field of a MAETT contain distinct elements with respect to the ACL2 function `equal`. In other words, two instructions i and j recorded in the MAETT MT satisfies (`equal` i j) iff they represent the same instruction.

This completes the description of our instruction representation. The INST structure stores the stages of the represented instruction, and this helps us to define properties for instructions. The pre-ISA and post-ISA states are also useful. The fact that instructions are recorded in program order is important in defining properties involving program order. Additionally, the MAETT records the information for speculative execution, branch prediction, internal exceptions and external interrupts. In the following sections, we define various functions and predicates that are defined using the MAETT intermediate abstraction.

7.4 Instruction Order

A MAETT records instructions in a list in the *trace* field. Formula $i \in_{\mathrm{MT}} MT$ denotes the fact that instruction i is one of the instructions recorded by MT. We define the relation $i \in_{\mathrm{MT}} MT$ as shown below. This functional definition tests whether i is a member of the list in MT.trace.

DEFINITION:
$i \in_{\mathrm{MT}} MT \stackrel{def}{=} i \in (MT.\mathrm{trace})$

If instruction i precedes another instruction j in program order, we write i **precedes** j **in** MT. The predicate member-in-order($elm1, elm2, lst$) in the following definition is true iff element *elm1* appears earlier than *elm2* in the list *lst*. This can be easily seen because ACL2 function `(member-equal` *elm list*`)`, which is printed here as $elm \in list$, returns the tail of *list* beginning with the first occurrence of *elm* if *elm* is a member of *list*.

DEFINITION:
member-in-order ($elm1$, $elm2$, lst) $\stackrel{def}{=} elm2 \in \mathrm{cdr}\,(elm1 \in lst)$

DEFINITION:
$i1$ **precedes** $i2$ **in** $MT \stackrel{def}{=}$ member-in-order ($i1$, $i2$, MT.trace)

From the definition of INST-in-order-p(i, j, MT), we can prove that the relation between i and j are anti-reflexive, anti-symmetric, transitive, and total. These theorems[2] can be proven by induction.

THEOREM: INST-in-order-p-identity
$(\mathrm{inv}\,(MT, MA) \wedge \text{MAETT-p}\,(MT) \wedge \text{MA-state-p}\,(MA) \wedge (i \in_{\mathrm{MT}} MT) \wedge \text{INST-p}\,(i))$
$\rightarrow (\neg\ (i$ **precedes** i **in** $MT))$

THEOREM: INST-in-order-antisymmetry
$(\mathrm{inv}\,(MT, MA) \wedge \text{MAETT-p}\,(MT) \wedge \text{MA-state-p}\,(MA) \wedge (i$ **precedes** j **in** $MT))$
$\rightarrow (\neg\ (j$ **precedes** i **in** $MT))$

THEOREM: INST-in-order-transitivity
$($ $\mathrm{inv}\,(MT, MA)$
$\wedge$ MAETT-p (MT)
$\wedge$ MA-state-p (MA)
$\wedge$ $(i$ **precedes** j **in** $MT)$
$\wedge$ $(j$ **precedes** k **in** $MT))$
$\rightarrow (i$ **precedes** k **in** $MT)$

[2]Several theorems proven in this chapter use predicate inv(MT, MA) as hypothesis. This predicate is our invariant condition that will be defined in the next chapter. In this chapter, consider inv(MT, MA) as the well-formedness predicate for the MAETT MT.

THEOREM: INST-in-order-p-total
$((i \in_{\mathrm{MT}} MT) \wedge (j \in_{\mathrm{MT}} MT) \wedge (\neg\ (j$ **precedes** i **in** $MT)) \wedge (i \neq j))$
$\rightarrow (i$ **precedes** j **in** $MT)$

The theorems shown in this section are some of the most basic properties that can be proven about instructions. The relations $i \in_{\mathrm{MT}} MT$ and i **precedes** j **in** MT are used in the definition of many predicates and theorems. We will see these relations be used in the definition of our invariant in the next chapter.

7.5 Specifying Instructions by Stages

The functions introduced in this section allow us to designate instructions with their stages or tags. For example, INST-at-stg(s, MT) designates the instruction at stage s, and INST-of-tag(tg, MT) designates the instruction with tag tg.

The function INST-at-stg(s, MT) returns the instruction at stage s recorded in MT. If more than one instruction is at stage s, it returns the first instruction in program order. If there is no instruction at stage s, it returns **nil**.

DEFINITION:
INST-at-stg-in-trace(s, *trace*)
$\stackrel{def}{=}$
if endp(*trace*) **then nil**
elseif s = (car(*trace*).stg) **then** car(*trace*)
else INST-at-stg-in-trace(s, cdr(*trace*))
fi

DEFINITION:
INST-at-stg(s, MT) $\stackrel{def}{=}$ INST-at-stg-in-trace(s, MT.trace)

The following predicates test whether any instructions are at a particular stage. The predicate no-INST-at-stg(s, MT) is true iff no instruction is at the stage s. The predicate uniq-INST-at-stg(stg, MT) is true iff there is exactly one at the stage s.

DEFINITION:
no-INST-at-stg-in-trace(s, *trace*)
$\stackrel{def}{=}$
if endp(*trace*) **then t**
elseif (car(*trace*).stg) = s **then nil**
else no-INST-at-stg-in-trace(s, cdr(*trace*))
fi

DEFINITION:
uniq-INST-at-stg-in-trace(s, *trace*)
$\stackrel{def}{=}$
if endp(*trace*) **then nil**
elseif (car(*trace*).stg) = s **then** no-INST-at-stg-in-trace(s, cdr(*trace*))
else uniq-INST-at-stg-in-trace(s, cdr(*trace*))
fi

DEFINITION:
no-INST-at-stg(s, MT) $\stackrel{def}{=}$ no-INST-at-stg-in-trace(s, MT.trace)

DEFINITION:
uniq-INST-at-stg(s, MT) $\stackrel{def}{=}$ uniq-INST-at-stg-in-trace(s, MT.trace)

We prove the basic properties of the functions and predicates defined above. THEOREM INST-stg-INST-at-stg proves that the stage of instruction defined as INST-at-stg(s, MT) is s, and THEOREM INST-at-stg-INST-stg shows that i is the only instruction at stage i.stg if uniq-INST-at-stg(i.stg, MT) is true

THEOREM: INST-stg-INST-at-stg
uniq-INST-at-stg(s, MT) $\rightarrow$ ((INST-at-stg(s, MT).stg) = s)

THEOREM: INST-at-stg-INST-stg
$((i \in_{\mathrm{MT}} MT) \wedge$ uniq-INST-at-stg(i.stg, MT))
$\rightarrow$ (INST-at-stg(i.stg, MT) = i)

We define similar functions and predicates that take a list of stages as an argument. The function INST-at-stgs(*s-list*, MT) returns the first instruction in program order whose stage is a member of a list of stages *s-list*. The predicate uniq-INST-at-stgs(*s-list*, MT) is true iff exactly one instruction is at one of the stages in *s-list*, and no-INST-at-stgs(*s-list*, MT) is true iff there is no such instruction.

Similar functions and predicates are defined for the tags of instructions. The FM9801 uses tags to identify instructions that produce operands. In fact, a tag designates an instructions uniquely. The function INST-of-tag(tg, MT) returns the instruction to which the tag tg is assigned. Predicates no-INST-of-tag(tg, MT) is true iff no instruction has the tag tg, while uniq-INST-of-tag(tg, MT) is true iff exactly one instruction recorded in MT has the tag tg. We show the definition of INST-of-tag(tg, MT) below.

DEFINITION:
INST-of-tag-in-trace (tg, *trace*)
$\stackrel{def}{=}$
if endp (*trace*) **then nil**
elseif ((car (*trace*).tag) = tg)
$\wedge$ dispatched-p (car (*trace*))
$\wedge$ ($\neg$ committed-p (car (*trace*))) **then** car (*trace*)
else INST-of-tag-in-trace (tg, cdr (*trace*))
fi

DEFINITION:
INST-of-tag (tg, MT) $\stackrel{def}{=}$ INST-of-tag-in-trace (tg, MT.trace)

We can prove theorems similar to those proven for INST-at-stg. THEOREM INST-tag-inst-of-tag shows that INST-of-tag(tg, MT) returns the instruction whose tag is tg, and THEOREM INST-of-tag-INST-tag indicates that i is the only instruction whose tag is i.tag.

THEOREM: INST-tag-INST-of-tag
((MAETT-p (MT) $\wedge$ ROB-index-p (tg)) $\wedge$ uniq-INST-of-tag (tg, MT))
$\rightarrow$ ((INST-of-tag (tg, MT).tag) = tg)

THEOREM: INST-of-tag-INST-tag
((inv (MT, MA) $\wedge$ MAETT-p (MT) $\wedge$ MA-state-p (MA))
$\wedge$ (INST-p (i) $\wedge$ ($i \in_{\mathrm{MT}} MT$))
$\wedge$ dispatched-p (i)
$\wedge$ ($\neg$ committed-p (i)))
$\rightarrow$ (INST-of-tag (i.tag, MT) = i)

7.6 Last Register Modifiers

The FM9801 uses Tomasulo's algorithm to forward the results of instructions to the instructions waiting for operand values in the reservation stations. An operand value is produced by the instruction that writes to the operand register immediately before the instruction that reads it. We define this operand-producing instruction as the last register modifier and use it in the verification of Tomasulo's algorithm.

We call an instruction that writes to register r as an *r-register modifier*. The *last r-register modifier before* instruction i is the last of all r-register modifiers that precedes i in program order. The *last r-register modifier in the reorder buffer* is the last of all r-register modifiers which are in the FIFO queue implemented by the reorder buffer. Since all dispatched and uncommitted instructions have an allocated entry in the reorder buffer, the last r-register modifier in the reorder buffer can be defined as the last dispatched but uncommitted r-register modifier.

For instance, in the following code fragment, i_0 and i_2 are R3-register modifiers, and i_2 is the last R3-register modifier before i_3. If instructions i_0, i_1, and i_2 are dispatched and not committed, and if i_3 is not dispatched, then i_1 is the last R2-register modifier in the reorder buffer.

```
i0:  R3 := ADD(R1,R5)
i1:  R2 := MUL(R1,R4)
i2:  R3 := ADD(R2,R6)
i3:  R2 := MUL(R3,R4)
```

The last register modifiers can be formalized using the MAETT abstraction. The predicate reg-modifier-p(r, i) holds when i is an r-register modifier, where r designates a general-purpose register. The function LRM-before(i, r, MT) defines the last r-register modifier before i. The predicate exist-LRM-before(i, r, MT) is true iff there exists the last r-register modifier before instruction i. The function

LRM-before(i, r, MT) returns **nil** if the last r-register modifier before i does not exist.

DEFINITION:
trace-exist-LRM-before-p $(i,\ r,\ trace)$
$\stackrel{def}{=}$
if endp $(trace)$ **then nil**
elseif car $(trace) = i$ **then nil**
elseif reg-modifier-p $(r,$ car $(trace))$ **then t**
else trace-exist-LRM-before-p $(i,\ r,$ cdr $(trace))$
fi

DEFINITION:
exist-LRM-before-p $(i,\ r,\ MT)$
$\stackrel{def}{=}$
trace-exist-LRM-before-p $(i,\ r,\ MT.$trace$)$

DEFINITION:
trace-LRM-before $(i,\ r,\ trace)$
$\stackrel{def}{=}$
if endp $(trace)$ **then nil**
elseif car $(trace) = i$ **then nil**
elseif reg-modifier-p $(r,$ car $(trace))$
$\wedge$ $(\neg$ trace-exist-LRM-before-p $(i,\ r,$ cdr $(trace)))$ **then** car $(trace)$
else trace-LRM-before $(i,\ r,$ cdr $(trace))$
fi

DEFINITION:
LRM-before $(i,\ r,\ MT) \stackrel{def}{=}$ trace-LRM-before $(i,\ r,\ MT.$trace$)$

The following three theorems show that LRM-before(i, r, MT) in fact defines the last r-register modifier before i. THEOREM reg-modifier-p-LRM-before states that LRM-before(i, r, MT) is an r-register modifier. THEOREM INST-in-order-LRM-before shows that the instruction defined by LRM-before(i, r, MT) precedes i in program order. THEOREM LRM-is-last implies that LRM-before(i, r, MT) is the last of all such r-register modifiers; if an r-register modifier j precedes instruction

i, either j precedes LRM-before(i, r, MT) or j itself is the last r-register modifier before i.

THEOREM: reg-modifier-p-LRM-before
exist-LRM-before-p $(i,\ r,\ MT) \rightarrow$ reg-modifier-p $(r,$ LRM-before $(i,\ r,\ MT))$

THEOREM: INST-in-order-LRM-before
$((i \in_{\mathrm{MT}} MT) \wedge$ exist-LRM-before-p $(i,\ r,\ MT))$
$\rightarrow$ (LRM-before $(i,\ r,\ MT)$ **precedes** i **in** MT)

THEOREM: LRM-is-last
((inv $(MT,\ MA) \wedge$ MAETT-p $(MT) \wedge$ MA-state-p $(MA))$
$\wedge$ $((i \in_{\mathrm{MT}} MT) \wedge$ INST-p $(i))$
$\wedge$ $((j \in_{\mathrm{MT}} MT) \wedge$ INST-p $(j))$
$\wedge$ reg-modifier-p $(r,\ j)$
$\wedge$ (j **precedes** i **in** MT)
$\wedge$ (LRM-before $(i,\ r,\ MT) \neq j$))
$\rightarrow$ (j **precedes** LRM-before $(i,\ r,\ MT)$ **in** MT)

Additionally, we can prove that the last r-register modifier before i produces the correct source operand value for instruction i. As introduced in Section 7.3.2, the function INST-dest-val(j) defines the result produced by instruction j. The ideal value of i's source operand register r is represented as read-reg$(r, i.$pre-ISA.RF$)$, because instruction i reads the value of register r in the pre-ISA state of i in the corresponding ISA execution. The following theorem shows that the last r-register modifier before i produces this ideal source operand value for i, if the last r-register modifier before i exists and if i is neither speculatively executed nor modified by self-modifying code.

THEOREM: INST-dest-val-LRM-before
((inv $(MT,\ MA) \wedge$ MAETT-p $(MT) \wedge$ MA-state-p $(MA) \wedge$ rname-p $(r))$
$\wedge$ (INST-p $(i) \wedge (i \in_{\mathrm{MT}} MT) \wedge ((i.\mathrm{specultv?}) \neq \mathtt{1}) \wedge ((i.\mathrm{modified?}) \neq \mathtt{1})$
$\wedge$ exist-LRM-before-p $(i,\ r,\ MT)$
$\wedge$ ($\neg$ committed-p (LRM-before $(i,\ r,\ MT)$))))
$\rightarrow$ (INST-dest-val (LRM-before $(i,\ r,\ MT)$) = read-reg $(r,\ (i.$pre-ISA$).$RF$)$)

We define similar predicates and functions for the last register modifier in the reorder buffer. The last r-register modifier in the reorder buffer is defined by the

function LRM-in-ROB(r, MT). The predicate exist-LRM-in-ROB-p(r, MT) tests the existence of the last r-register modifier in the reorder buffer.

The theory presented so far is about the instructions that modify general-purpose registers. We can develop an almost identical theory for special registers. For instance, we define the function LSRM-before(i, sr, MT) which defines the last sr-register modifier before i, where sr designates a special register. The predicate exist-LSRM-before-p(i, sr, MT) is true iff the last sr-register modifier exists.

In the next chapter, we discuss the invariant properties defined using our MAETT abstraction just introduced. For example, we use the INST representation to define the conditions that each instruction should satisfy at particular pipeline stages. Another example is the properties related to data-dependencies, which can be represented as recursive functions because instructions are recorded in a MAETT in program order.

Chapter 8

Definition and Verification of Invariant Properties

A number of properties hold invariantly during the MA execution of the FM9801. In this section, we discuss the definition and verification of such invariant properties. Defining and proving the invariant properties are the basis of the proof for our correctness criterion in the next chapter. First, we discuss the definition of invariant properties in Subsection 8.1. Then, we briefly discuss their verification in Section 8.2.

8.1 Definition of the Invariant Condition

8.1.1 Overview

Currently, specifications of large hardware designs, such as microprocessor models, cannot be automatically verified. We need to break down the large verification problem into a number of subproblems, each of which is small enough to handle with verification tools.

We are particularly interested in writing an invariant condition in such a way that it can help us to decompose the verification problem of an entire microprocessor

into the verification of a number of simpler machine properties, which can be verified independently. Using the MAETT intermediate abstraction that represents the executed instructions with the INST data-structure, we define various properties that should hold during the correct operation of our microprocessor. We define an invariant condition as the conjunction of such properties defined on the MAETT. We verify individual properties independently of each other to reduce the problem size. The correctness criterion is deduced from the validity of the invariant conditions as we will discuss in the next chapter.

As discussed in Chapter 6, self-modifying code may cause a pipelined MA to behave differently from its ISA specification. Thus our invariant condition may not hold if self-modifying code is completely executed. However, our invariant condition holds during the speculative execution of modified instructions, because speculatively executed instructions should not have any side-effects on the programmer visible states as they will be abandoned later.

We define the predicate MT-CMI-p(MT), which implies that some committed instruction recorded in MAETT MT are modified by self-modifying code. The actual definition of MT-CMI-p(MT) can be found in Appendix D.

Lemma 1 *(Committed Modified Instruction)*

$$\text{MT-CMI-p}(MT) \rightarrow \exists i \in_{\text{MT}} MT \ \{i.\text{modified?} = 1 \wedge \text{committed-p}(i)\}$$

From the definition of MT-step and MT-CMI-p, we can prove the following lemma:

Lemma 2 *Suppose MA is a microarchitectural state and MT is its MAETT abstraction state. Let MT' be the next MAETT state,* MT-step($MT, MA, sigs$)*, after one step of MA execution with input signals sigs. Then,*

$$\text{MT-CMI-p}(MT) \rightarrow \text{MT-CMI-p}(MT') \ .$$

In other words, commitment of modified instruction cannot be undone.

We define our invariant inv(MT, MA) as follows:

Definition 1 *Let* Π *be the set of properties shown in Table 8.1.*

$$\text{inv}(MT, MA) \stackrel{def}{=} \bigwedge_{P \in \Pi} P(MT, MA)$$

The predicate inv(MT, MA) satisfies the following two lemmas.

Theorem 2

$$\text{flushed-p}(MA) \rightarrow \text{inv}(\text{MT-init}(MA), MA)$$

Theorem 3 *Let* MA' *be the next MA state* MA-step(MA, *sigs*) *and* MT' *be the next MAETT state* MT-step(MT, MA, *sigs*). *Then,*

$$\text{inv}(MT, MA) \rightarrow \text{inv}(MT', MA') \vee \text{MT-CMI-p}(MT')$$

Theorem 2 states that the initial flushed state MA and its MAETT abstraction MT-init(MA) satisfy our invariant. Theorem 3 states that our invariant condition inv(MT, MA) is an *invariant under the constraint* ¬MT-CMI-p(MT) [LL90]; that is, inv(MT, MA) is invariantly true as long as no modified instructions are committed.[1] Intuitively speaking, if our invariant is true for the current state MA and its MAETT abstraction MT, then our invariant will be true for the next states MA' and MT' or some modified instruction will have been committed.

From Lemma 2, Theorems 2 and 3, we can prove the following theorem by induction.

Theorem 4 *Suppose* MA_0 *to be an initial MA state. Let*

$$\begin{aligned} MT_0 &= \text{MT-init}(MA_0) \\ MA_n &= \text{MA-stepn}(MA_0, \textit{sig-list}, n) \\ MT_n &= \text{MT-stepn}(MT_0, MA_0, \textit{sig-list}, n) \end{aligned}$$

[1] According to Lamport and Lynch, $P(s)$ is an invariant under the constraint under $C(s)$ iff $P(s) \wedge C(s) \rightarrow P(s') \vee \neg C(s')$, where s and s' is the current and next state of the system. This is equivalent to $P(s) \rightarrow P(s') \vee \neg C(s')$ if $\neg C(s) \rightarrow \neg C(s')$, which is true for the ¬MT-CMI-p(MT).

Sec.	Property Name	Brief Description
8.1.2	**weak-inv:**	A well-formedness predicate for a MAETT.
8.1.3	**in-order-dispatch-commit-p:**	Instructions are dispatched and committed in sequential execution order.
8.1.4	**in-order-DQ-p:**	The dispatch queue is a FIFO queue.
8.1.5	**in-order-ROB-p:**	The reorder buffer is a FIFO queue.
8.1.6	**in-order-LSU-inst-p:**	Certain instruction orders are preserved for memory access instructions in the load-store unit.
8.1.7	**no-stage-conflict:**	No structural conflict at pipeline stages.
8.1.8	**no-tag-conflict:**	No structural conflict in the reorder buffer.
8.1.9	**correct-speculation-p:**	Instructions are speculatively executed if they follow a mispredicted branch, exception, or context switching instruction.
8.1.10	**no-specultv-commit-p:**	No speculatively executed instruction commits.
8.1.11	**correct-exintr-p:**	An externally interrupted instruction retires immediately.
8.1.12	**MT-INST-inv:**	Valid intermediate data values in the pipeline.
8.1.13	**consistent-RS-p:**	Reservation stations keep track of the tag of the operand-producing instructions.
8.1.14	**consistent-reg-tbl-p:**	The register reference table keeps track of the last general register modifying instruction.
8.1.14	**consistent-sreg-tbl-p:**	The register reference table keeps track of the last special register modifying instruction.
8.1.15	**pc-match-p:**	Correct state of the program counter.
8.1.15	**SRF-match-p:**	Correct state of the special register file.
8.1.15	**RF-match-p:**	Correct state of the general register file.
8.1.15	**mem-match-p:**	Correct state of the memory.
8.1.16	**consistent-MA-p:**	Some properties for an MA state.
8.1.16	**misc-inv:**	The conjunction of miscellaneous invariants.

Table 8.1: List of properties that should hold during the normal execution of the FM9801. We define Π to be the set of all properties listed here. This table also shows the subsection in which each property is discussed.

Then,

$$\text{flushed-p}(MA_0) \rightarrow \text{inv}(MT_n, MA_n) \vee \text{MT-CMI-p}(MT_n)$$

This theorem states that any MA state reachable from a flushed state satisfies our invariant condition unless some modified instructions are committed.

In the following subsections, we will discuss the definition of properties listed in Table 8.1. The table shows the number of the subsection in which each property is discussed. The verification of the invariant condition is discussed in the second half of this chapter.

8.1.2 Weak Invariants

A predicate weak-inv(MT) is a well-formedness predicate for a MAETT. Its definition is given below. Conditions used in the definition of weak-inv(MT) check whether consistent values are in the fields *ID*, *modified?*, *first-modified?*, *pre-ISA* and *post-ISA* of the INST representations of the recorded instructions. It also checks no two INST representations in MAETT MT are identical.

DEFINITION:
weak-INV (MT)
$\stackrel{def}{=}$
  MT-new-ID-distinct-p (MT)
$\wedge$ MT-distinct-IDs-p (MT)
$\wedge$ MT-distinct-inst-p (MT)
$\wedge$ ISA-step-chain-p (MT)
$\wedge$ correct-modified-flgs-p (MT)
$\wedge$ correct-modified-first (MT)

Since inv(MT, MA) is an invariant under constraint $\neg$MT-CMI-p(MT), it is guaranteed to hold only when a self-modifying program is not executed. However, the condition weak-INV(MT) is an invariant condition without any constraints. Thus, the following theorem is provable.

THEOREM: weak-INV-step
(MAETT-p (MT)
$\wedge$ MA-state-p (MA)
$\wedge$ MA-input-p ($sigs$)
$\wedge$ weak-INV (MT))
$\rightarrow$ weak-INV (MT-step (MT, MA, $sigs$))

8.1.3 Order of Instruction Fetch, Dispatch and Commit

The FM9801 fetches, dispatches and commits instructions in order. Since the MAETT records instructions in program order, we can define a simple predicate of a MAETT that tests this property.

Suppose the trace of a MAETT MT is MT.trace $= (i_0\ i_1 \cdots i_{n-1})$. Since instructions $i_0\ \cdots\ i_{n-1}$ are in program order, the following three conditions must hold:

1. i_{n-1} is the only instruction that can be at the IFU stage.
2. If i_j is dispatched and i_k is not dispatched, then $j < k$.
3. If i_j is committed and i_k is not committed, then $j < k$.

The first condition must hold because instructions are fetched in order and the IFU stage is the first stage for every instruction. The second and the third conditions hold because instructions are dispatched and committed in order.

The predicate in-order-dispatch-commit-p(MT) is true iff the MAETT MT satisfies these three conditions. The definition uses two predicates in order to represent the conditions 2 and 3. The predicates no-dispatched-INST-p($trace$) and no-committed-INST-p($trace$) are true iff list $trace$ contains no element representing a dispatched or committed instruction, respectively.

DEFINITION:
no-dispatched-INST-p ($trace$)
$\stackrel{def}{=}$
if endp ($trace$) **then t**

else ($\neg$ dispatched-p (car (*trace*))) $\wedge$ no-dispatched-INST-p (cdr (*trace*))
fi

DEFINITION:
no-commit-INST-p (*trace*)
$\stackrel{def}{=}$
if endp (*trace*) **then t**
else ($\neg$ committed-p (car (*trace*))) $\wedge$ no-commit-INST-p (cdr (*trace*))
fi

DEFINITION:
in-order-trace-p (*trace*)
$\stackrel{def}{=}$
if endp (*trace*) **then t**
else **if** IFU-stg-p (car (*trace*).stg) **then** endp (cdr (*trace*))
elseif $\neg$ dispatched-p (car (*trace*)) **then** no-dispatched-INST-p (cdr (*trace*))
elseif $\neg$ committed-p (car (*trace*)) **then** no-commit-INST-p (cdr (*trace*))
else t
fi
$\wedge$ in-order-trace-p (cdr (*trace*))
fi

DEFINITION:
in-order-dispatch-commit-p (*MT*) $\stackrel{def}{=}$ in-order-trace-p (*MT*.trace)

From the definition of the property in-order-dispatch-commit-p, we can derive the following theorems. THEOREM INST-in-order-dispatch-undispatch states that instruction i precedes instruction j in program order if i is dispatched and j is not. Similarly, THEOREM INST-in-order-commit-uncommit states that i precedes j in program order if i is committed and j is not. These theorems are proven using the fact that the invariant condition inv(MT, MA) implies all properties in Table 8.1.

THEOREM: INST-in-order-dispatched-undispatched
((inv (*MT*, *MA*) $\wedge$ MAETT-p (*MT*) $\wedge$ MA-state-p (*MA*))
$\wedge$ (($i \in_{\mathrm{MT}} MT$) $\wedge$ ($j \in_{\mathrm{MT}} MT$))
$\wedge$ dispatched-p (i)
$\wedge$ ($\neg$ dispatched-p (j)))
$\rightarrow$ (i **precedes** j **in** MT)

THEOREM: INST-in-order-commit-uncommit
((inv (*MT*, *MA*) $\wedge$ MAETT-p (*MT*) $\wedge$ MA-state-p (*MA*))
$\wedge$ (($i \in_{\mathrm{MT}} MT$) $\wedge$ ($j \in_{\mathrm{MT}} MT$))

$\wedge$ committed-p (i)
$\wedge$ ($\neg$ committed-p (j)))
$\rightarrow$ (i **precedes** j **in** MT)

8.1.4 Order of Instructions in the Dispatch Queue

The dispatch queue implements a FIFO buffer with four entries. The instructions in the dispatch queue are the stages `'(DQ 0)`, `'(DQ 1)`, `'(DQ 2)`, and `'(DQ 3)`. The instruction at stage `'(DQ 0)` is the first in program order, and the instructions at stages `'(DQ 1)`, `'(DQ 2)`, and `'(DQ 3)` follow in that order. The predicate in-order-DQ-p(MT) in Table 8.1 tests whether this order is observed by the instructions in the dispatch queue. Like the definition of in-order-dispatch-commit-p(MT), the definition of in-order-DQ-p(MT) relies on the fact that MAETT records instructions in program order.

The following theorem shows the idea behind the definition of in-order-DQ-p. In the theorem, the function DQ-stg-idx(s) returns the index to the dispatch queue entry, given that s is one of the dispatch queue stages shown above. For instance, DQ-stg-idx(`'(DQ 1)`) = `1`. Thus, the theorem states that, if instructions i and j are both in the dispatch queue and i precedes j in program order, the instruction i is ahead of j in the dispatch queue. The theorem is directly derived from the definition of in-order-DQ-p.

THEOREM: DQ-stg-index-monotonic
(inv (MT, MA)
$\wedge$ MAETT-p (MT)
$\wedge$ MA-state-p (MA)
$\wedge$ DQ-stg-p (i.stg)
$\wedge$ DQ-stg-p (j.stg)
$\wedge$ (i **precedes** j **in** MT))
$\rightarrow$ (DQ-stg-idx (i.stg) $<$ DQ-stg-idx (j.stg))

8.1.5 Order of Instructions in the Reorder Buffer

The FM9801 implements out-of-order completion of instructions. After the execution is completed, the instructions are reordered in the reorder buffer and committed in order. The predicate in-order-ROB-p(MT) tests whether reorder buffer correctly records the order of all dispatched but uncommitted instructions so that it can recover the original program order.

Unlike the dispatch queue, the reorder buffer implements a circular buffer using two indices and a flag. Figure 8.1 shows the relation between the indices and the flag. The index *head* points to the oldest instruction in the reorder buffer, and the index *tail* points to the entry following the newest instruction. The shaded part of the buffer contains valid instructions. Whenever an instruction is added to and removed from the buffer, the indices *tail* and *head* are incremented, respectively. When either index reaches the bottom of the buffer, it is reset to the top of the buffer and the wrap-around flag, *flg*, is toggled. Initially when the buffer is empty, $flg = 0$ and $head = tail$.

We define a relation $rix1 <_{\mathrm{tag}} rix2$ **in** MT, which holds when the instruction at entry $rix1$ is ahead of another instruction at entry $rix2$ in the FIFO queue implemented by the reorder buffer. In the ACL2 logic, this relation between the indices is defined with the predicate `(tag-in-order` *rix1 rix2 MT*`)`, whose name is derived from the fact that the FM9801 uses the reorder buffer indices as the tags for Tomasulo's algorithm. The reorder buffer store the current values of *flg*, *head*, and *tail* in its fields. The MAETT abstraction records these values in its fields *ROB-flg*, *ROB-head*, and *ROB-tail*, respectively. Thus, the relation $rix1 <_{\mathrm{tag}} rix2$ **in** MT is defined as follows.

DEFINITION:
rix1 $<_{\mathrm{tag}}$ *rix2* **in** MT
$\stackrel{def}{=}$
if (MT.ROB-flg) = 1
then $(((MT.\text{ROB-head}) \leq rix1) \wedge (rix2 < (MT.\text{ROB-tail})))$

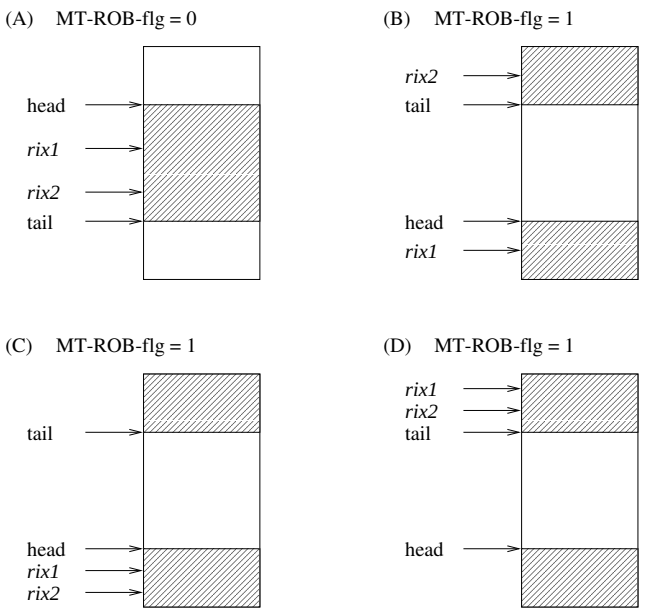

Figure 8.1: Order of Instructions in the Reorder Buffer

$$\vee \ (((MT.\text{ROB-head}) \leq rix1) \wedge (rix1 < rix2))$$
$$\vee \ ((rix1 < rix2) \wedge (rix2 < (MT.\text{ROB-tail})))$$

else $rix1 < rix2$

fi

This definition first checks the value of flg recorded in MT.ROB-flg. If it is not equal to 1, the relation between the pointers is as shown in (A) in Fig. 8.1 and $rix1 < rix2$ implies $rix1 <_{\text{MT}} rix2$ **in** MT. If MT.ROB-flg $= 1$, pointers should satisfy one of the three relations shown in (B), (C), and (D)

Predicate in-order-ROB-p(MT) in Table 8.1 determines whether the reorder buffer records instructions in program order. The following theorem intuitively shows the idea behind the definition of in-order-ROB-p. As mentioned earlier, i.tag designates the index of the reorder buffer entry which is assigned to instruction i. According to the theorem, for any dispatched but uncommitted instructions i and j, i precedes j in program order if i.tag $<_{\text{MT}}$ j.tag **in** MT.

THEOREM: INST-in-order-INST-of-tag-if-tag-in-order

$$
\begin{array}{l}
(\quad (\text{inv}\,(MT,\, MA) \wedge \text{MAETT-p}\,(MT) \wedge \text{MA-state-p}\,(MA)) \\
\quad \wedge\ (\text{INST-p}\,(i) \wedge (i \in_{\text{MT}}\ MT) \wedge (\neg\ \text{committed-p}\,(i)) \wedge \text{dispatched-p}\,(i)) \\
\quad \wedge\ (\text{INST-p}\,(j) \wedge (j \in_{\text{MT}}\ MT) \wedge (\neg\ \text{committed-p}\,(j)) \wedge \text{dispatched-p}\,(j)) \\
\quad \wedge\ ((i.\text{tag}) <_{\text{tag}} (j.\text{tag})\ \textbf{in}\ MT)) \\
\rightarrow (i\ \textbf{precedes}\ j\ \textbf{in}\ MT)
\end{array}
$$

8.1.6 Orders of Load and Store Instructions

The order of instructions is critical for the correct implementation of memory operations. Out-of-order execution of load and store instructions can cause hazardous results, which may violate the condition of precise exceptions. Moreover, the order of instructions must be known to implement load-bypassing and load-forwarding, which were explained in Subsection 5.3.9.

The predicate in-order-LSU-inst-p(MT, MA) determines whether the following five conditions are met for the load and store instructions:

1. The reservation station holds instructions in program order.
2. Instructions are issued from the reservation station in order.
3. The write buffer holds instructions in program order.
4. The write buffer releases store instructions in order.
5. The order between the store instructions in the write buffer and the load instruction in the read buffer are correctly recorded.

In the following definition of in-order-LSU-INST-p(MT, MA), each condition is represented as a conjunct. The predicates used in this definition are defined similarly to the predicates discussed in the last few sections.

DEFINITION:

$$
\begin{array}{l}
\text{in-order-LSU-INST-p}\,(MT,\, MA) \\
\stackrel{def}{=} \\
\quad \text{in-order-LSU-RS-p}\,(MT.\text{trace},\, MA)
\end{array}
$$

$\wedge$ in-order-LSU-issue-p (MT.trace)
$\wedge$ in-order-WB-trace-p (MT.trace)
$\wedge$ in-order-WB-retire-p (MT.trace)
$\wedge$ in-order-load-store-p (MT, MA)

8.1.7 Absence of Stage Conflicts

The following two subsections describe methods for checking structural conflicts in the pipelined machines. Typically, structural conflicts occur when multiple instructions need the same structural resource. Stage conflicts are one kind of structural conflicts that might occur at pipeline latches. In the FM9801, pipeline latches are state holding devices such as the IFU, the dispatch queue entries and the reservation stations. Each latch can contain at most one instruction. If multiple instructions try to occupy the same pipeline latch, a structural conflict occurs. The predicate no-stage-conflict(MT, MA) tests whether stage conflicts exist in state MA.

DEFINITION:
no-stage-conflict (MT, MA)
$\stackrel{def}{=}$
no-IFU-stg-conflict (MT, MA)
$\wedge$ no-DQ-stg-conflict (MT, MA)
$\wedge$ no-IU-stg-conflict (MT, MA)
$\wedge$ no-MU-stg-conflict (MT, MA)
$\wedge$ no-LSU-stg-conflict (MT, MA)
$\wedge$ no-BU-stg-conflict (MT, MA)

In this definition, the six predicates check whether multiple instructions occupy a single latch in the IFU, the dispatch queue, the integer unit, the multiply unit, the load-store unit, and the branch unit, respectively. For example, the definition of no-IFU-stg-conflict(MT, MA) is given below. When the busy flag of the IFU unit, (MA.IFU).valid?, is set to 1, there should be exactly one instruction at the `'(IFU)` stage. If the flag is set to 0, no instruction should be at the `'(IFU)` stage. Similarly, we can define the conflict-free property of other pipeline stages.

DEFINITION:
no-IFU-stg-conflict (MT, MA)
$\stackrel{def}{=}$
if ((MA.IFU).valid?) = `1` **then** uniq-INST-at-stg (`'(IFU)`, MT)
else no-INST-at-stg (`'(IFU)`, MT)
fi

8.1.8 Absence of Conflicts in the Reorder Buffer

The reorder buffer also satisfies a conflict-free property; no more than one instruction can occupy the same reorder buffer entry. The definition of the predicate no-tag-conflict(MT, MA), which tests this property, is given below.

DEFINITION:
no-tag-conflict-at (idx, MT, MA)
$\stackrel{def}{=}$
if (nth-robe (idx, MA.ROB).valid?) = `1` **then** uniq-INST-of-tag (idx, MT)
else no-INST-of-tag (idx, MT)
fi

DEFINITION:
no-tag-conflict-under (idx, MT, MA)
$\stackrel{def}{=}$
if $idx \simeq$ `0` **then t**
else no-tag-conflict-at ($idx - 1$, MT, MA)
$\wedge$ no-tag-conflict-under ($idx - 1$, MT, MA)
fi

DEFINITION:
no-tag-conflict (MT, MA) $\stackrel{def}{=}$ no-tag-conflict-under (`8`, MT, MA)

The predicate no-tag-conflict-at(idx, MT, MA) implies that no conflict occurs at the reorder buffer entry indexed by idx. The function nth-robe(idx, MA.ROB) returns the state of the idx'th entry of reorder buffer in state MA. If its busy flag, nth-robe(idx, MA.ROB).valid?, is set to 1, exactly one instruction is stored in the reorder buffer entry. The predicate no-tag-conflict(MT, MA) is true if and only if no-tag-conflict-at(idx, MA.ROB) holds for all eight reorder buffer entries.

As mentioned earlier, the FM9801 uses the index to the allocated reorder buffer entry as the tag for an instruction. Thus, the conflict-free property in the reorder buffer implies the uniqueness of the tags. The following theorem implies that different tags are assigned to each dispatched but uncommitted instruction. The proof uses the definition of the predicate no-tag-conflict.

THEOREM: tag-identity
$$
\begin{array}{l}
\quad (\quad (\mathrm{inv}\,(MT,\, MA) \wedge \text{MAETT-p}\,(MT) \wedge \text{MA-state-p}\,(MA)) \\
\qquad \wedge\ (\text{INST-p}\,(i) \wedge (i \in_{\mathrm{MT}}\ MT) \wedge \text{dispatched-p}\,(i) \wedge (\neg\ \text{committed-p}\,(i))) \\
\qquad \wedge\ (\text{INST-p}\,(j) \wedge (j \in_{\mathrm{MT}}\ MT) \wedge \text{dispatched-p}\,(j) \wedge (\neg\ \text{committed-p}\,(j)))) \\
\rightarrow (((i.\mathrm{tag}) = (j.\mathrm{tag})) \leftrightarrow (i = j))
\end{array}
$$

8.1.9 Speculatively Executed Instructions

The predicate correct-speculation-p(MT) tests whether the MAETT MT correctly records which instructions are speculatively executed. In Subsection 7.3.3, we generalized the concept of the speculative execution; the FM9801 starts speculative execution from an uncommitted instruction that causes either a mispredicted branch, an exception, or context synchronization. If instruction i causes speculative execution in this sense, i satisfies INST-start-specultv?$(i, MT) = 1$. All subsequent instructions of i are speculatively executed. This relation is illustrated with the following theorem.

THEOREM: INST-in-order-p-INST-start-specultv
$$
\begin{array}{l}
\quad (\quad \mathrm{inv}\,(MT,\, MA) \\
\qquad \wedge\ \text{MAETT-p}\,(MT) \\
\qquad \wedge\ \text{MA-state-p}\,(MA) \\
\qquad \wedge\ (\text{INST-start-specultv?}\,(i) = \mathtt{1}) \\
\qquad \wedge\ (i\ \textbf{precedes}\ j\ \textbf{in}\ MT)) \\
\rightarrow ((j.\mathrm{specultv?}) = \mathtt{1})
\end{array}
$$

According to this theorem, if i is an uncommitted instruction that starts speculative execution, a subsequent instruction j is speculatively executed. The INST representation has a field *specultv?* which is set to 1 when the represented instruction is speculatively executed. Thus, i.specultv? = 1 implies that i is speculatively

executed. The predicate correct-speculation-p(MT) tests whether the MAETT MT correctly records the instructions which are speculatively executed in this sense.

8.1.10 Abandoning Speculatively Executed Instructions

According to our definition of speculative execution, all speculatively executed instructions must be abandoned before they are committed. This property is represented by the predicate no-specultv-commit-p(MT), which is true if no speculatively executed instruction recorded in MT is committed. The following theorem can be proven from the definition of no-specultv-commit-p(MT). If instruction i is committed, it should not be executed speculatively.

THEOREM: not-INST-specultv-INST-in-if-committed
$$
\begin{array}{l}
(\quad (\text{inv}\,(MT,\, MA) \wedge \text{MAETT-p}\,(MT) \wedge \text{MA-state-p}\,(MA)) \\
\quad \wedge\ (\text{INST-p}\,(i) \wedge (i \in_{\text{MT}}\ MT)) \\
\quad \wedge\ \text{committed-p}\,(i)) \\
\rightarrow ((i.\text{specultv?}) = \texttt{0})
\end{array}
$$

8.1.11 Stage of Interrupted Instructions

The predicate correct-exintr-p(MT) tests whether all externally interrupted instructions are retired. As discussed in Section 5.3.8, the FM9801 goes through a synchronization process to handle an external interrupt. When an external interrupt signal is received, the processor halts further dispatch of instructions, completes the execution of dispatched instructions, interrupts the first undispatched instruction, and abandons the subsequent instructions. In our MAETT modeling, the interrupted instruction goes to the `'(retire)` stage at the end of the synchronization process. It is also at this time the *exintr?* field of the INST representation of the interrupted instruction is set to 1. Thus, any interrupted instruction i satisfying i.exintr? $= 1$ is at the `'(retire)` stage. The following theorem, proven from the definition of correct-exintr-p(MT), shows this relation.

THEOREM: INST-exintr-INST-in-if-not-retired
$$
\begin{array}{l}
\quad (\quad (\text{inv}\,(MT,\ MA) \wedge \text{MAETT-p}\,(MT) \wedge \text{MA-state-p}\,(MA)) \\
\quad\ \wedge\ (\text{INST-p}\,(i) \wedge (i \in_{\text{MT}}\ MT)) \\
\quad\ \wedge\ (\neg\ \text{retire-stg-p}\,(i.\text{stg}))) \\
\rightarrow ((i.\text{exintr?}) = \texttt{0})
\end{array}
$$

8.1.12 Correctness of Intermediate Values

In this subsection, we define the correctness of the pipeline intermediate values with the predicate MT-INST-inv(MT, MA). In the FM9801, a single instruction can go through as many as 11 stages before it is retired. The intermediate results of an instruction are stored in pipeline latches. As an instruction advances to the next stage, the intermediate results for the next latch are generated from the intermediate results stored in the previous latch. Eventually, the instruction produces the final result, which is stored into the register file or the memory. The correctness of the final results depends on the correctness of the intermediate values.

The definition of MT-INST-inv(MT, MA) determines whether every instruction i recorded in the MAETT MT satisfies the predicate INST-inv(i, MA). Intuitively speaking, INST-inv(i, MA) is true iff the intermediate results of i are correctly stored in the corresponding latch in state MA.

DEFINITION:
trace-INST-inv ($trace$, MA)
$\stackrel{def}{=}$
if endp ($trace$) **then t**
else INST-inv (car ($trace$), MA) $\wedge$ trace-INST-inv (cdr ($trace$), MA)
fi

DEFINITION:
MT-INST-inv (MT, MA) $\stackrel{def}{=}$ trace-INST-inv (MT.trace, MA)

The predicate INST-inv(i, MA) is defined by case analysis on the stage of instruction i. For example, IFU-INST-inv(i, MA) defines the correct intermediate values of the instruction i at the `'(IFU)` stage. Other predicates represent the correct intermediate values for multiple stages; for example, DQ-INST-inv(i, MA)

defines the correct intermediate values at stages `'(DQ 0)`, `'(DQ 1)`, `'(DQ 2)`, and `'(DQ 3)`.

DEFINITION:
INST-inv(i, MA)
$\stackrel{def}{=}$
if IFU-stg-p(i.stg) **then** IFU-INST-inv(i, MA)
elseif DQ-stg-p(i.stg) **then** DQ-INST-inv(i, MA)
elseif execute-stg-p(i.stg) **then** execute-INST-inv(i, MA)
elseif complete-stg-p(i.stg) **then** complete-INST-inv(i, MA)
elseif commit-stg-p(i.stg) **then** commit-INST-inv(i, MA)
else t
fi

In order to illustrate the idea of how we define the intermediate values for each stage, the definition of IFU-inst-inv is presented below. In this definition, we assume that i is an instruction at the `'(IFU)` stage. As shown in Figure 5.4, the IFU has four fields *valid?*, *pc*, *excpt*, and *word*. The predicate IFU-inst-inv tests that these fields contain the ideal intermediate values.

DEFINITION:
IFU-INST-inv(i, MA)
$\stackrel{def}{=}$

$$
\begin{array}{ll}
 & ((MA.\text{IFU}).\text{valid?}) = 1) \\
\wedge & (\quad (((i.\text{specultv?}) \neq 1) \wedge (((i.\text{modified?}) \neq 1) \vee ((i.\text{first-modified?}) = 1))) \\
 & \quad \rightarrow (((MA.\text{IFU}).\text{pc}) = ((i.\text{pre-ISA}).\text{pc}))) \\
\wedge & (\quad (((i.\text{specultv?}) \neq 1) \wedge ((i.\text{modified?}) \neq 1)) \\
 & \quad \rightarrow (((MA.\text{IFU}).\text{excpt}) = \text{INST-excpt-flags}(i))) \\
\wedge & (\quad (((i.\text{specultv?}) \neq 1) \wedge ((i.\text{modified?}) \neq 1) \wedge (\neg\ \text{INST-fetch-error-detected-p}(i))) \\
 & \quad \rightarrow (((MA.\text{IFU}).\text{word}) = \text{INST-word}(i))) \\
\wedge & (\quad (((i.\text{specultv?}) \neq 1) \wedge ((i.\text{modified?}) \neq 1) \wedge \text{INST-fetch-error-detected-p}(i)) \\
 & \quad \rightarrow (((MA.\text{IFU}).\text{word}) = 0))
\end{array}
$$

The busy flag *valid?* of the IFU should be set to 1, because the IFU is occupied by the instruction i. The field *pc* should contain the address of the stored instruction i. The address of the instruction i is expressed as i.pre-ISA.pc, which is the program counter value in the pre-ISA state of i. This is the address from which the ISA

fetches i for the corresponding execution. The *pc* field value in the IFU should be equal to this ideal address of instruction i, if i is not speculatively executed and it is not modified by self-modifying code. If the instruction i is executed speculatively, the processor may be executing the instruction differently from the way the ISA executes the same instruction. Thus, the actual intermediate values may not be equal to the ideal values defined in terms of the ISA execution. Similarly, modified instructions may not be executed correctly with respect to the ISA execution.

The predicate IFU-inst-inv also tests the intermediate values at the field *excpt* and *word*. The ideal exception status which should be recorded in the *excpt* field is defined by the function INST-excpt-flags(i), and the ideal value of the *word* field is defined by INST-word(i). Functions INST-excpt-flags(i) and INST-word(i) were introduced in Subsection 7.3.2

Correctness of intermediate values at other stages are similarly defined. The entire definition of INST-inv(i, MA) includes 161 equalities, each of which relates the actual value in the microarchitectural state MA and the ideal intermediate value for instruction i. The predicate MT-INST-inv has the largest definition of all the properties shown in Table 8.1. This makes its verification challenging. In Subsection 8.2.2, we will revisit this property and discuss its verification.

8.1.13 Correct Tags in Reservation Stations

In the FM9801 execution core, the results of instruction execution are forwarded through the CDB to the reservation stations where instructions wait for their source operands, as described in Section 5.3. Tags are used to identify the instructions that produce the source operand values. The correctness of the tags stored in the reservation stations is critical for the verification of forwarded data values. The predicate consistent-RS-p(MT, MA) tests whether tags stored in the reservation stations are correct.

In Section 7.6, we defined the last r-register modifier before i as the instruction that writes to general-purpose register r before the instruction i. If an instruction i uses register r as its source operand, the last r-register modifier before i is the instruction that produces the source operand value. We proved this with THEOREM INST-dest-val-LRM-before in Section 7.6. Therefore, the reservation stations must keep the tags of the last register modifiers if the source operands are not ready.

The predicate consistent-RS-p(MT, MA) in Table 8.1 tests whether all four reservation stations in the FM9801 keep the correct tag of the last register modifiers. As an example, we discuss the correct tag values in the reservation station attached to the multiply unit. A multiply instruction needs the values of two source operand registers which are specified by the instruction fields *ra* and *rb*. As described in Subsection 5.3.4, the fields of the reservation station entry, *ready1?*, records whether the *ra* register value is ready, and if it is not, the field *src1* stores the tag of the instruction that will produce the new value of the *ra* register. Suppose instruction i is at reservation station entry 0, then the correctness of the tag value is given by the theorem shown below. The value of its *src1* field is expressed as MA.MU.RS0.src1. The ideal tag value is the tag of the last r-register modifier before i, where r the operand register specified by the *ra* instruction field. Using the function introduced in Subsection 7.3.2, r is represented as INST-ra(i). In the following theorem, the actual tag value in the *src1* field of the reservation station is proven to be equal to this ideal tag value if the *ready1?* flag is not set, and i is neither speculatively executed nor modified. This is one of the properties tested by consistent-RS-p(MT, MA).

THEOREM: MU-RS0-src1-INST-tag-LRM
(inv (MT, MA)
$\wedge$ ((i.stg) = '(MU RS0))
$\wedge$ (((MA.MU).RS0).ready1?) $\neq$ 1)
$\wedge$ ((i.specultv?) $\neq$ 1)
$\wedge$ ((i.modified?) $\neq$ 1)
$\wedge$ MAETT-p (MT)

$\wedge$ MA-state-p (MA)
$\wedge$ INST-p (i)
$\wedge$ ($i \in_{\mathrm{MT}} MT$))
$\rightarrow$ (((((MA.MU).RS0).src1) = (LRM-before (i, INST-ra (i), MT).tag))

8.1.14 Tags in Register Reference Table

Register reference tables keep track of the instructions that will produce the newest value of the registers. The predicate consistent-reg-tbl-p(MT, MA) in Table 8.1 tests whether the register reference table for the general-purpose register file is working correctly. Separately, we define a predicate consistent-sreg-tbl-p(MT, MA) which tests the register reference table for the special register file. Since the definition of consistent-sreg-tbl-p(MT, MA) is very similar to consistent-reg-tbl-p(MT, MA), we only discuss the latter.

The register reference table keeps the tag of the most recently dispatched instruction that will modify a specific register. This instruction can be characterized as the last r-register modifier in the reorder buffer, which is defined in Section 7.6,

The predicate consistent-reg-tbl-p(MT, MA) tests whether every register r satisfies consistent-reg-ref-p(r, MT, MA), which defines the correct values of the *wait* and *tag* fields in the corresponding register reference table entry. The *wait* field for register r is set to 1 iff the last r-register modifier in the reorder buffer exists, because the register r contains an old value and is waiting for the new value produced by the last register modifier. In such a case, the *tag* field should contain the tag of the last r-register modifier in the reorder buffer.

DEFINITION:
consistent-reg-ref-p (r, MT, MA)
$\stackrel{def}{=}$
if (MT-specultv-at-dispatch? (MT) = 1) $\vee$ (MT-modified-at-dispatch? (MT) = 1)
then t
elseif (reg-tbl-nth (r, (MA.DQ).reg-tbl).wait?) = 1
then exist-LRM-in-ROB-p (r, MT)
$\wedge$ ((LRM-in-ROB (r, MT).tag) = (reg-tbl-nth (r, (MA.DQ).reg-tbl).tag))
else $\neg$ exist-LRM-in-ROB-p (r, MT)

fi

DEFINITION:
consistent-reg-tbl-under $(r,\ MT,\ MA)$
$\stackrel{def}{=}$
if $r \simeq$ `0` **then t**
else consistent-reg-ref-p $(r - 1,\ MT,\ MA)$
$\wedge$ consistent-reg-tbl-under $(r - 1,\ MT,\ MA)$
fi

DEFINITION:
consistent-reg-tbl-p $(MT,\ MA) \stackrel{def}{=}$ consistent-reg-tbl-under $(\mathtt{8},\ MT,\ MA)$

The tags in the register reference table are used to identify the instructions producing the source operand values for dispatched instructions. The MA copies the tag in the register reference table to a reservation station when an instruction is dispatched. The following theorem shows that the last r-register modifier in the reorder buffer is the last r-register modifier before i when i is at the head of the dispatch queue entry. This implies that the tag stored in the register reference table correctly specifies the last register modifier before the dispatched instruction, which produces the source operand value.

THEOREM: INST-dest-val-LRM-in-ROB
(inv $(MT,\ MA)$
$\wedge$ $((i.\mathrm{stg}) =$ `'(DQ 0)`$)$
$\wedge$ exist-LRM-in-ROB-p $(rname,\ MT)$
$\wedge$ MAETT-p (MT)
$\wedge$ MA-state-p (MA)
$\wedge$ INST-p (i)
$\wedge$ $(i \in_{\mathrm{MT}} MT)$)
$\rightarrow$ (LRM-in-ROB $(rname,\ MT)$ = LRM-before $(i,\ rname,\ MT)$)

8.1.15 Correct States of Programmer Visible Components

Predicates pc-match-p(MT, MA), RF-match-p(MT, MA), SRF-match-p(MT, MA), and mem-match-p(MT, MA), listed in Table 8.1, check whether the program counter, the general-purpose register file, the special register file, and the memory, respectively, are in the correct states. What is common in these predicates is that they

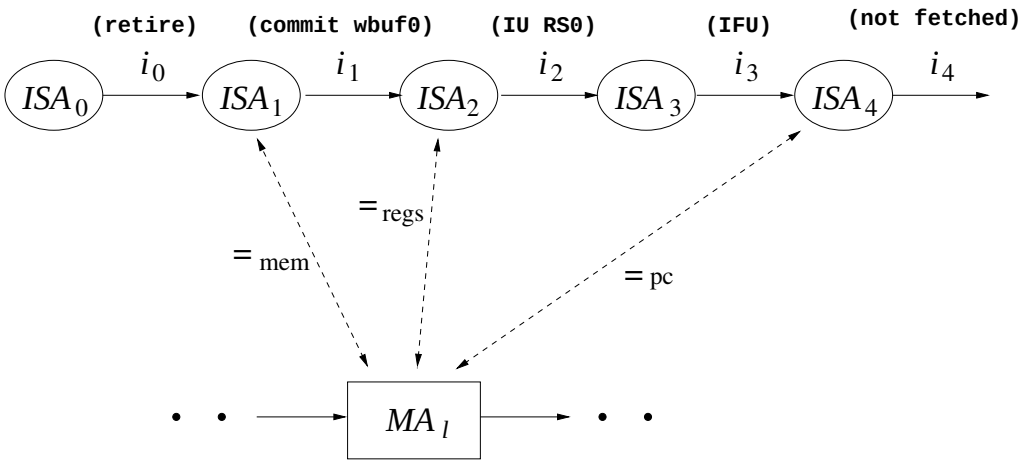

Figure 8.2: Relation between the states of the programmer visible components in the ISA and the MA.

define the ideal component states using the MAETT abstraction and compare them with the actual component states. Since the definition of the RF-match-p and SRF-match-p are almost identical, we skip the discussion on SRF-match-p and explain the remaining three conditions in this subsection.

An example relation between the programmer-visible component states in the ISA and the MA is shown in Fig. 8.2. Let us assume that we have been executing instructions $i_0, i_1, i_2, i_3, \ldots$. Let ISA_k be i_k.pre-ISA, which is also equal to i_{k-1}.post-ISA if $k > 0$. Suppose that, in the MA state MA_l, stages of instructions i_0, i_1, i_2, and i_3 are `'(retire)`, `'(commit wbuf0)`, `'(IU RS0)`, and `'(IFU)`, respectively, and instructions after i_4 have not been fetched. Also, we assume that none of the instructions are speculatively executed nor modified by self-modifying code. As we describe below, dashed-arrows show the correspondence between the states of the program counter, the general-purpose register file, and the memory in the ISA and the MA.

The program counter in MA_l should hold the address of the instruction i_4 because it is the next instruction to be fetched. In the ISA execution, this

address is held by the program counter in the state ISA_4, from which the ISA fetches instruction i_4. In general, the program counter in the post-ISA state of the most recently fetched instruction has the address of the instruction to be fetched next.

Results of instructions are written back to the register file when the instructions are committed. Since instructions i_0 and i_1 are committed but i_2 and i_3 are not, the register file in MA_l records the results of i_0 and i_1, just like the ISA state ISA_2 does. That means that the register file state ISA_2 is the ideal state of the register file in MA_l. In general, the ideal register file state in the MA is defined as that of the post-ISA state of the most recently committed instruction, since instructions are committed in order.

The memory is updated when a store instruction is released from the write buffer and retired. Since i_0 is retired but i_1 is not, the memory state in the MA_l is as if it were in the ISA state immediately after executing i_0. Therefore the memory states in the MA_l and ISA_1 should be equal. The ideal memory state for an MA state is the pre-ISA state of the first instruction that has not been retired.

As illustrated in the example above, the ideal component states can be found in the pre-ISA and post-ISA states of instructions. Since the MAETT records such pre-ISA and post-ISA states, we can define the ideal component states as functions that take a MAETT as the sole argument.

DEFINITION:
trace-pc (*trace*, *pre-pc*)
$\stackrel{def}{=}$
if endp (*trace*) **then** *pre-pc*
else trace-pc (cdr (*trace*), (car (*trace*).post-ISA).pc)
fi

DEFINITION:
MT-pc (MT) $\stackrel{def}{=}$ trace-pc (MT.trace, (MT.Init-ISA).pc)

DEFINITION:
pc-match-p (MT, MA)

$\stackrel{def}{=}$
(b-nor (MT-in-specultv? (MT), MT-self-modify? (MT)) = 1)
$\rightarrow$ (MT-pc (MT) = (MA.pc))

The function MT-pc(MT) defines the ideal program counter value for the current MA state. The function MT-pc(MT) returns the program counter value in the post-ISA state of the last instruction recorded the MAETT MT, because it is the most recently fetched instruction. If no instructions are recorded in the MAETT, it simply returns the program counter value of the initial ISA state, MT.init-ISA.

When the MA is speculatively executing instructions, it may be fetching instructions from a wrong address and the program counter may not be correct with respect to the ISA execution. Similarly, if modified instructions are executed by the MA, the program counter may be incorrect. The predicate pc-match-p(MT, MA) checks the equivalence between the ideal program counter value MT-pc(MT) and the actual program counter value in state MA except these cases.

Similarly, the RF-match-p and mem-match-p check whether the register file and the memory are in the ideal states.

DEFINITION:
trace-RF (*trace*, RF)
$\stackrel{def}{=}$
if endp (*trace*) **then** RF
elseif $\neg$ committed-p (car (*trace*)) **then** RF
else trace-RF (cdr (*trace*), (car (*trace*).post-ISA).RF)
fi

DEFINITION:
MT-RF (MT) $\stackrel{def}{=}$ trace-RF (MT.trace, (MT.Init-ISA).RF)

DEFINITION:
RF-match-p (MT, MA) $\stackrel{def}{=}$ MT-RF (MT) = (MA.RF)

DEFINITION:
trace-mem (*trace*, *mem*)
$\stackrel{def}{=}$
if endp (*trace*) **then** *mem*

elseif ¬ retire-stg-p (car (*trace*).stg) **then** *mem*
else trace-mem (cdr (*trace*), (car (*trace*).post-ISA).mem)
fi

DEFINITION:
MT-mem (MT) $\stackrel{def}{=}$ trace-mem (MT.trace, (MT.Init-ISA).mem)

DEFINITION:
mem-match-p (MT, MA) $\stackrel{def}{=}$ MT-mem (MT) = (MA.mem)

The function MT-RF(MT) defines the ideal register file state as the post-ISA state of the last committed instruction. The function MT-mem(MT) defines the ideal memory state similarly. Unlike the predicate pc-match-p(MT, MA), the predicates RF-match-p(MT, MA) and mem-match-p(MT, MA) do not check whether instructions are speculatively executed or modified by self-modifying code. This is because the register file and the memory are always in the correct state, since speculatively executed instructions are never committed. The register file and the memory are not contaminated by self-modifying code either, because the constraint ¬MT-CMI-p(MT) of our invariant inv(MT, MA) implies that no modified instructions are committed.

8.1.16 Other Invariant Conditions

In addition to the properties discussed so far, we needed several more properties to complete the FM9801 verification. We defined these remaining properties with two predicates consistent-MA-p and misc-inv in Table 8.1.

The predicate consistent-MA-p(MA) collects properties that can be easily defined on the MA state without its MAETT abstraction. In the definition given below, consistent-MA-p(MA) checks whether the dispatch queue, the reorder buffer, and the load-store unit satisfies certain conditions. For instance, the control vector in the dispatch queue should satisfy the constraints that guarantee each instruction is dispatched to only one reservation station.

DEFINITION:
consistent-MA-p (MA)
$\stackrel{def}{=}$
  consistent-DQ-cntlv-p (MA.DQ)
$\wedge$ consistent-ROB-p (MA.ROB)
$\wedge$ consistent-LSU-p (MA.LSU)

The predicate misc-inv(MT, MA) checks other relations between an MA state and its MAETT. For instance, misc-inv checks whether *ROB-flg*, *ROB-head*, and *ROB-tail* of the MAETT MT correctly record the actual wrap-around flag, the head and tail pointers in the reorder buffer. It also checks if the *DQ-len* field of the MAETT records the correct number of instructions in the dispatch queue.

DEFINITION:
misc-inv (MT, MA)
$\stackrel{def}{=}$
  ((MA.ROB).flg) = (MT.ROB-flg))
$\wedge$ ((MA.ROB).head) = (MT.ROB-head))
$\wedge$ ((MA.ROB).tail) = (MT.ROB-tail))
$\wedge$ ((MT.DQ-len) $\leq$ 4)
$\wedge$ correct-entries-in-DQ-p (MT, MA)

This completes the description of each property used in the definition of our invariant condition. In each subsection, we have discussed one or more properties listed in Table 8.1. In the next section, we discuss the verification of the invariant condition defined in this section.

8.2 Verification of the Invariant Condition

8.2.1 Overview

We defined our invariant as a conjunction of all the properties in Table 8.1. Our invariant condition inv(MT, MA) can be represented as $\bigwedge_{P \in \Pi} P(MT, MA)$ where Π is the set of predicates shown in Table 8.1. The proof of Theorem 2 is a rather

straightforward base case proof. In order to prove Theorem 3, it suffices to show the following formula for every $P \in \Pi$:

$$\text{inv}(MT, MA) \wedge \neg\text{MT-CMI-p}(MT') \rightarrow P(MT', MA'), \tag{8.1}$$

where $MA' = \text{MA-step}(MA, sigs)$ and $MT' = \text{MT-step}(MT, MA, sigs)$. In other words, we need to prove every property P in Π for the next state pair MA' and MT', assuming that $\text{inv}(MT, MA)$ holds for the current state pair MA and MT and that no modified instructions commit in this step.

The verification of Formula (8.1) for the individual properties can be done independently. When we verify a property P, we can concentrate our computational resource and human effort on the microarchitectural components related to the property P, neglecting the rest of the machine design. For instance, when we verify the property in-order-DQ-p which states that the dispatch queue implements a FIFO queue, we can concentrate our effort on the microarchitectural components related to the dispatch queue. When we verify the property in-order-ROB-p, which checks whether the reorder buffer implements a FIFO queue, we switch our attention to the microarchitectural components related to the reorder buffer, and disregard the rest of the design.

The following theorem proves Formula (8.1) for the case where $P(MT, MA)$ is RF-match-p(MT, MA). Additional assumptions are type predicates.

```
THEOREM: RF-match-p-preserved
  (   (MAETT-p (MT) ∧ MA-state-p (MA) ∧ MA-input-p (sigs))
   ∧  inv (MT, MA)
   ∧  (¬ MT-CMI-p (MT-step (MT, MA, sigs))))
→ RF-match-p (MT-step (MT, MA, sigs), MA-step (MA, sigs))
```

Using the ACL2 theorem prover, we proved similar theorems corresponding to Formula (8.1) for all properties in Table 8.1. Theorem 2 and 3 are then proved in the following theorems.

```
THEOREM: inv-initial-MT
(MA-state-p (MA) ∧ (MA-flushed? (MA) = 1)) → inv (init-MT (MA), MA)
```

THEOREM: inv-step
((MAETT-p (*MT*) ∧ MA-state-p (*MA*) ∧ MA-input-p (*sigs*))
∧ inv (*MT*, *MA*)
∧ (¬ MT-CMI-p (MT-step (*MT*, *MA*, *sigs*))))
→ inv (MT-step (*MT*, *MA*, *sigs*), MA-step (*MA*, *sigs*))

In the following subsection, we will look closely into a few verification problems encountered during the proof of these theorems. We discuss the verification of intermediate values in Subsection 8.2.2, the verification of forwarded data in Tomasulo's algorithm in Subsection 8.2.3, and the verification of memory access operations in Subsection 8.2.4.

8.2.2 Verification of Intermediate Values

As discussed in Subsection 8.1.12, the property MT-INST-inv defines the correct intermediate values in the pipeline. Formula (8.1) for MI-INST-inv is proven in the theorem shown below. This theorem implies that all intermediate values in the next machine state MA-step($MA, sigs$) are correct.

THEOREM: MT-INST-inv-preserved
((MAETT-p (*MT*) ∧ MA-state-p (*MA*) ∧ MA-input-p (*sigs*))
∧ inv (*MT*, *MA*)
∧ (¬ MT-CMI-p (MT-step (*MT*, *MA*, *sigs*))))
→ MT-INST-inv (MT-step (*MT*, *MA*, *sigs*), MA-step (*MA*, *sigs*))

The proof by induction decomposes the theorem into subgoals which imply the correctness of intermediate values for individual instructions. The following theorem shows that the intermediate values for instruction i is correct in the next MA state MA-step($MA, sigs$), given that the invariant condition inv(MT, MA) holds for the current state. The hypothesis MT-no-jmp-exintr-before($i, MT, MA, sigs$) implies that instruction i is not abandoned due to speculative execution or an external interrupt. The hypothesis INST-exintr-now?($i, MA, sigs$) $\neq 1$ implies that i itself is not externally interrupted.

THEOREM: INST-inv-step-INST
((MAETT-p (*MT*) ∧ MA-state-p (*MA*) ∧ MA-input-p (*sigs*))
∧ (INST-p (*i*) ∧ (*i* ∈MT *MT*))
∧ inv (*MT*, *MA*)
∧ (¬ MT-CMI-p (MT-step (*MT*, *MA*, *sigs*)))
∧ MT-no-jmp-exintr-before (*i*, *MT*, *MA*, *sigs*)
∧ (INST-exintr-now? (*i*, *MA*, *sigs*) ≠ 1))
→ INST-inv (step-INST (*i*, *MT*, *MA*, *sigs*), MA-step (*MA*, *sigs*))

The proof of this theorem is carried out by case analysis on the stages of the instruction i in the current machine state MA and next machine state MA'. If the instruction is at the pipeline stage S in the current machine state MA and it moves to the stage S' in the next machine state MA', we assume the correctness of the intermediate values at stage S in MA and show the validity of the intermediate values at stage S' in MA'.

However, manually constructing proofs for individual cases is time consuming, because the definition of MI-INST-inv(i, MT) contains 161 equalities between the actual values in the MA machine state and the ideal intermediate values. We automated some of the proofs of these equalities.

The key to improve the proof efficiency is the use of ACL2 rewriting rules. Each rewriting rule is defined in such a way that ACL2 terms representing intermediate values in the MA state are rewritten to irreducible terms involving the INST representation of instructions.

For example, we can prove the following theorem

THEOREM: IFU-word-INST-word
(inv (*MT*, *MA*)
∧ MAETT-p (*MT*)
∧ MA-state-p (*MA*)
∧ (*i* ∈MT *MT*)
∧ IFU-stg-p (*i*.stg)
∧ ((*i*.specultv?) ≠ 1)
∧ ((*i*.modified?) ≠ 1))
→ (((*MA*.IFU).word) = INST-word (*i*))

This theorem states that the intermediate value stored in the *word* field of the IFU is equal to INST-word(i), which represents the ideal instruction word of i. The ACL2 theorem prover uses this theorem as a rewriting rule that converts (MA.IFU).word to INST-word(i).

We define similar rewriting rules for each field in all the pipeline latches. These rewriting rules rewrite expressions representing the actual intermediate values in the machine state to the expressions representing the ideal values defined on the INST representation of instructions. Most of the ideal values are defined with the functions introduced in the Subsection 7.3.2.

With these rewriting rules, the ACL2 theorem prover can automatically prove the validity of many intermediate values while proving Theorem INST-inv-step-INST. Of all the equalities appearing in the definition of MT-INST-inv, only those that cannot be verified automatically are attacked with human interaction. This lowered the cost of verifying the validity of intermediate values, allowing us to completely verify THEOREM MT-INST-inv-preserved.

8.2.3 Correctness of Forwarded Data Values

In this subsection, we consider how to verify the correctness of the data-forwarding in the pipeline. As discussed in Chapter 5, Tomasulo's algorithm is used to forward the results from the execution units through the CDB to the reservation stations, where instructions wait for their operands. Proving the correctness of these forwarded values is one of the most challenging problems in the verification of the FM9801.

As discussed in Section 7.6, if instruction i is waiting for the value of the source operand register r, the reservation station should keep the tag of the last r-register modifier before i. When the tag in the reservation station matches the tag on the bus *CDB-tag*, the reservation station reads the value from the bus *CDB-val* and uses it as an operand.

As an example, we consider verifying the data-forwarding to the reservation station attached to the multiply unit. We need three critical theorems to prove the correctness of the forwarded data value. First, THEOREM INST-dest-val-LRM-before in Section 7.6 states that the result produced by the last r-register modifier before i is the correct value of operand register r of instruction i. Second, THEOREM MU-RS0-src1-INST-tag-LRM in Subsection 8.1.13 states that the *src1* field of the reservation station stores the correct tag of the last register modifier before i. Third, THEOREM CDB-val-INST-dest-val*, which is given below, proves the following fact: if j is the instruction whose tag is on the bus *CDB-tag*, and if the bus *CDB-ready?* is set to 1, then the bus *CDB-val* carries the result of j.

THEOREM: CDB-val-INST-dest-val*
let j **be** INST-of-tag(CDB-tag(MA), MT)
in
((inv(MT, MA) ∧ MAETT-p(MT) ∧ MA-state-p(MA))
∧ INST-writeback-p(j)
∧ ((j.specultv?) ≠ 1)
∧ ((j.modified?) ≠ 1)
∧ (¬ INST-excpt-detected-p(j))
∧ (CDB-ready?(MA) = 1))
→ (CDB-val(MA) = INST-dest-val(j))

Using these theorems, we can prove the following theorem. The function INST-src-val1(i) specifies the correct value of the operand register specified by the *ra* instruction field.

THEOREM: CDB-val-INST-src-val1-if-CDB-ready-for-MU-RS0*
((inv(MT, MA) ∧ MAETT-p(MT) ∧ MA-state-p(MA))
∧ (INST-p(i) ∧ (i ∈$_{\mathrm{MT}}$ MT))
∧ ((i.stg) = '(MU RS0))
∧ ((i.specultv?) ≠ 1)
∧ ((i.modified?) ≠ 1)
∧ (CDB-ready?(MA) = 1)
∧ (CDB-tag(MA) = (((MA.MU).RS0).src1))
∧ ((((MA.MU).RS0).ready1?) ≠ 1))
→ (CDB-val(MA) = INST-src-val1(i))

Sketch of Proof: Let us denote INST-ra(i) as r and assume the hypotheses of the

theorem. From THEOREM MU-RS0-src1-INST-tag-LRM in Subsection 8.1.13 and the hypothesis CDB-tag(MA) = MA.MU.RS0.src, we have:

$$\text{CDB-tag}(MA) = \text{LRM-before}(i, r, MT).\text{tag}.$$

In other words, the tag on the CDB designates the last r-register modifier before i. Using this equality and THEOREM INST-of-tag-INST-tag in Section 7.5, instruction j in THEOREM CDB-val-INST-dest-val* is calculated as follows:

$$\begin{array}{lcll} j & = & \text{INST-of-tag}(\text{CDB-val}(MA), MT) & \{\text{Def. of } j\} \\ & = & \text{INST-of-tag}(\text{LRM-before}(i, r, MT).\text{tag}, MT) & \{\text{Equality shown above}\} \\ & = & \text{LRM-before}(i, r, MT) & \{\text{INST-of-tag-INST-tag}\} \end{array}$$

Using THEOREM INST-dest-val-LRM-before in Section 7.6, we can show the conclusion of the theorem.

$$\begin{array}{lcll} & & \text{CDB-val}(MA) & \\ & = & \text{INST-dest-val}(\text{LRM-before}(i, r, MT)) & \{\text{CDB-val-INST-dest-val*}\} \\ & = & \text{read-reg}(r, i.\text{pre-ISA.RF}) & \{\text{INST-dest-val-LRM-before}\} \\ & = & \text{INST-src-val1}(i) & \{\text{Def. of INST-src-val1}\} \end{array}$$

□

This theorem says that the forwarded value CDB-val(MA) is the correct operand register value of i. Similarly, we can prove the correctness of data values forwarded to other reservation stations.

8.2.4 Verification of Load-Forwarding and Load-Bypassing

The load-store unit of the FM9801 implements load-forwarding and load-bypassing as discussed in Section 5.3.9. Load-bypassing executes load and store instructions out of order, giving priorities to load instructions. Load-forwarding uses the value which will be stored in the memory as the result of a future load instruction. In

either case, the behaviors of the load instructions depend on the preceding store instructions.

In order to verify these techniques used in the load-store unit, we define the last memory modifiers in the same spirit as we defined the last register modifiers. We call the instruction that modifies the memory at address ad a *memory modifier* at address ad. The *last memory modifier at* address ad *before* instruction i is the last of all memory modifiers at address ad that precede instruction i in program order. The function LMM-before(i, ad, MT) defines the last memory modifier at address ad before i, and the predicate exist-LMM-before-p(i, ad, MT) tests its existence. These function and predicate are defined in the same way as we formalized the last register modifiers. Additionally we define exist-non-retired-LMM-before-p(i, ad, MT) which is true iff there exists a last memory modifier at address a before instruction i and it is not retired.

One important lemma for the correctness of load-bypassing is shown below. The function INST-src-val3(j) defines the operand value of a store instruction j that will be written to the memory. The theorem states that the value written to the memory by the last memory modifier at ad before i is the correct memory value at address ad for the instruction i.

THEOREM: INST-src-val3-LMM-before
((inv (MT, MA) $\wedge$ MAETT-p (MT) $\wedge$ MA-state-p (MA) $\wedge$ addr-p (ad))
$\wedge$ (($i \in_{\mathrm{MT}} MT$) $\wedge$ INST-p (i))
$\wedge$ ((i.specultv?) $\neq$ 1)
$\wedge$ ((i.modified?) $\neq$ 1)
$\wedge$ execute-stg-p (i.stg)
$\wedge$ exist-LMM-before-p (i, ad, MT)
$\wedge$ ($\neg$ retire-stg-p (LMM-before (i, ad, MT).stg)))
$\rightarrow$ (INST-src-val3 (LMM-before (i, ad, MT)) = read-mem (ad, (i.pre-ISA).mem))

Using the theorem above we proved the correctness of the load-forwarding in the following theorem. The function LSU-forward-wbuf(MA.LSU) defines the load-forwarded value in the FM9801, and LSU-address-match?(MA.LSU) is set to 1

when load-forwarding is taking place. The following theorem states that the load-forwarded value is the correct destination value of i if i is the load instruction in the read buffer.

THEOREM: LSU-forward-wbuf-INST-dest-val
$$
\begin{array}{l}
(\quad (\text{inv}\,(MT,\, MA) \wedge \text{MAETT-p}\,(MT) \wedge \text{MA-state-p}\,(MA) \wedge \text{MA-input-p}\,(sigs)) \\
\quad \wedge\ ((i \in_{\text{MT}} MT) \wedge \text{INST-p}\,(i)) \\
\quad \wedge\ ((i.\text{specultv?}) \neq \texttt{1}) \\
\quad \wedge\ ((i.\text{modified?}) \neq \texttt{1}) \\
\quad \wedge\ ((i.\text{stg}) = \texttt{'(LSU rbuf)}) \\
\quad \wedge\ (\text{release-rbuf?}\,(MA.\text{LSU},\, MA,\, sigs) = \texttt{1}) \\
\quad \wedge\ (\text{LSU-address-match?}\,(MA.\text{LSU}) = \texttt{1})) \\
\rightarrow (\text{LSU-forward-wbuf}\,(MA.\text{LSU}) = \text{INST-dest-val}\,(i))
\end{array}
$$

The correctness of load-bypassing is also proved using the concept of memory modifiers. The first theorem declares that, if the hardware line LSU-address-match? is not set, then the last memory memory modifier before the load instruction i does not exist or it is retired. The second theorem states that the current memory state MA.mem contains the same memory value at address ad as the memory state in the pre-ISA state of i, when the last memory modifier does not exist or it is retired.

THEOREM: not-exist-non-retired-LMM-before-p-if-not-address-match
$$
\begin{array}{l}
(\quad (\text{inv}\,(MT,\, MA) \wedge \text{MA-state-p}\,(MA)) \\
\quad \wedge\ (\text{INST-p}\,(i) \wedge (i \in_{\text{MT}} MT)) \\
\quad \wedge\ ((i.\text{specultv?}) \neq \texttt{1}) \\
\quad \wedge\ ((i.\text{modified?}) \neq \texttt{1}) \\
\quad \wedge\ ((i.\text{stg}) = \texttt{'(LSU rbuf)}) \\
\quad \wedge\ (\text{LSU-address-match?}\,(MA.\text{LSU}) \neq \texttt{1})) \\
\rightarrow (\neg\ \text{exist-non-retired-LMM-before-p}\,(i,\, \text{INST-load-addr}\,(i),\, MT))
\end{array}
$$

THEOREM: read-mem-when-no-active-mem-modifier-before
$$
\begin{array}{l}
(\quad (\text{inv}\,(MT,\, MA) \wedge \text{MAETT-p}\,(MT) \wedge \text{MA-state-p}\,(MA)) \\
\quad \wedge\ \text{addr-p}\,(ad) \\
\quad \wedge\ (i \in_{\text{MT}} MT) \\
\quad \wedge\ \text{INST-p}\,(i) \\
\quad \wedge\ (\neg\ \text{retire-stg-p}\,(i.\text{stg})) \\
\quad \wedge\ (\neg\ \text{exist-non-retired-LMM-before-p}\,(i,\, ad,\, MT))) \\
\rightarrow (\text{read-mem}\,(ad,\, MA.\text{mem}) = \text{read-mem}\,(ad,\, (i.\text{pre-ISA}).\text{mem}))
\end{array}
$$

Combining the two theorems above, we obtain the correctness of the load-bypassing in the following theorem. It states that the memory in the current state

MA holds the correct value for a load instruction i, if no address match is detected. Therefore, we can execute the load instruction in the current state without waiting for the completion of preceding store instructions.

THEOREM: read-mem-INST-load-addr-INST-dest-val
$$
\begin{array}{l}
(\quad (\text{inv}\,(MT,\, MA) \wedge \text{MAETT-p}\,(MT) \wedge \text{MA-state-p}\,(MA)) \\
\quad \wedge\ ((i \in_{\text{MT}} MT) \wedge \text{INST-p}\,(i)) \\
\quad \wedge\ ((i.\text{specultv?}) \neq \texttt{1}) \\
\quad \wedge\ ((i.\text{modified?}) \neq \texttt{1}) \\
\quad \wedge\ ((i.\text{stg}) = \texttt{'(LSU rbuf)}) \\
\quad \wedge\ (\text{LSU-address-match?}\,(MA.\text{LSU}) \neq \texttt{1})) \\
\rightarrow (\text{read-mem}\,(\text{INST-load-addr}\,(i),\, MA.\text{mem}) = \text{INST-dest-val}\,(i))
\end{array}
$$

8.2.5 Summary

The verification of the invariant conditions was the most time-consuming part of the FM9801 project. In order to verify our invariant condition whose definition involves 190 functions and predicates, we needed to prove 1878 theorem. Prior to this proof, we build additional 1232 "shared lemmas" about the MA and the MAETT abstraction, which were used as a basic ACL2 rule library during the verification of the invariant. Since verifying invariant conditions requires profound analysis on the components in the MA design, it is natural that this phase of verification takes the largest portion of the verification effort. Furthermore, all of the designs faults found in the original design of the FM9801 were detected during the verification of our invariant, as discussed in Chapter 10. The next chapter discusses the proof of the correctness correctness criterion, which puts together the various properties discussed in this chapter.

Chapter 9

Proof of Correctness Criterion

In the last chapter, we looked into our invariant properties of the FM9801. By verifying the invariant properties, we checked whether each microarchitectural component works correctly. Our remaining verification task is combining these results together to form the proof of our correctness criterion

A rough argument of the proof can be given with Figure 9.1. Suppose we are trying to verify our correctness criterion for an MA state transition sequence from the initial flushed state MA_0 to the final flushed state MA_n. Each MA state MA_k has the corresponding MAETT state MT_k, and the invariant condition inv(MT_k, MA_k) holds unless a self-modifying program is executed. Each MAETT records the completed and in-flight instructions in the corresponding MA state. Particularly, the MAETT MT_n corresponding to the final state MA_n records all instructions i_0 through i_{m-1} which were executed during the MA transition sequence. The MAETT also records the pre-ISA and post-ISA states of each instruction. In other words, the MAETT MT_n records the ISA state transition sequence from ISA_0 to ISA_m, where ISA_k is the pre-ISA state of instruction i_k and ISA_{k+1} is the post-ISA state of i_k. The initial ISA state ISA_0 is assumed to be equal to the projection, proj(MA_0), of the initial MA state MA_0. In order to prove our criterion, we need to show that

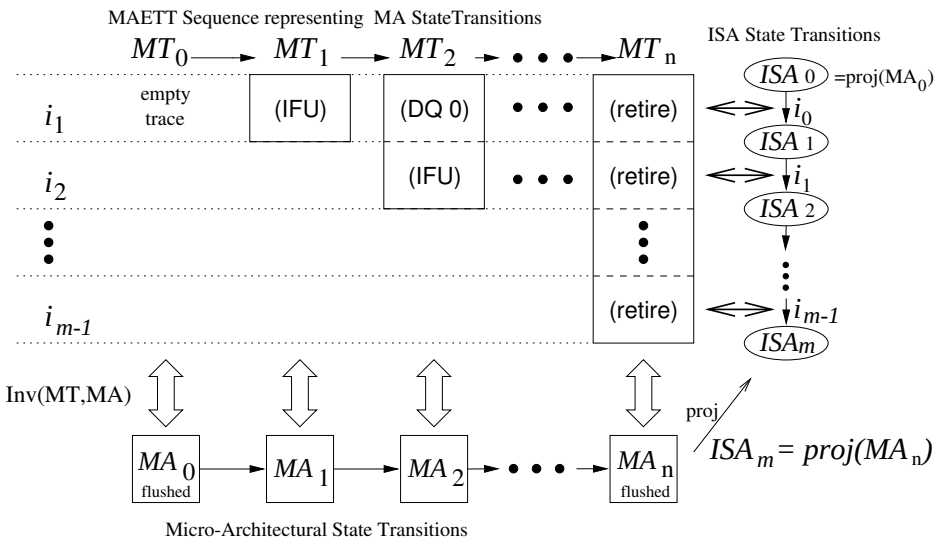

Figure 9.1: Relation between ISA, MA, and MAETT sequences.

the final ISA state ISA_m is equal to proj(MA_n).

The MAETT records the current pipeline stage of each instruction. For instance, MT_1 records that instruction i is at the `'(IFU)` stage in state MA_1. Since the final state MA_n is flushed, all instructions recorded in MT_n are retired. This implies that MA_n looks as if it had just finished the execution of all instructions i_0 through i_{m-1}. By using a simple analysis on MT_n with the invariant condition inv(MT_n, MA_n), we can show that the state of all programmer visible components in MA_n are equal to those in ISA_m. Thus, we can prove that proj(MA_n) = ISA_m and the commutative diagram holds.

In the rest of this chapter, we look into the proof more carefully. First, we introduce several functions to be used in this section. The function MT-len(MT) returns the length of the list in the field *trace* of MAETT MT. This is the number of instructions recorded in MT. The predicate MT-all-retired-p(MT) is true when all instructions recorded in MT are retired. The function MT-exintr-lst(MT) extracts the value of the *exintr?* field of each INST representation of instructions. For exam-

ple, let us consider a MAETT MT such that MT.trace $= (i_0\ \ i_1)$. If instructions i_0 and i_1 satisfy i_0.exintr? $= 0$ and i_1.exintr? $= 1$, then MT-exintr-lst(MT) = `'(0 1)`.

DEFINITION:
MT-len (MT) $\stackrel{def}{=}$ len (MT.trace)

DEFINITION:
trace-all-retired (*trace*)
$\stackrel{def}{=}$
if endp (*trace*) **then t**
else retire-stg-p (car (*trace*).stg) $\wedge$ trace-all-retired (cdr (*trace*))
fi

DEFINITION:
MT-all-retired-p (MT) $\stackrel{def}{=}$ trace-all-retired (MT.trace)

DEFINITION:
trace-exintr-lst (*trace*)
$\stackrel{def}{=}$
if endp (*trace*) **then nil**
else cons (ISA-input (car (*trace*).exintr?), trace-exintr-lst (cdr (*trace*)))
fi

DEFINITION:
MT-exintr-lst (MT) $\stackrel{def}{=}$ trace-exintr-lst (MT.trace)

The following several lemmas are needed in the final proof of the correctness criterion. Suppose MA is a flushed MA state and MT is its MAETT. Then no instructions are in the pipeline in the state MA and all instructions recorded in MT are retired.

Lemma 3 *Suppose MA is a microarchitectural state and MT is its MAETT abstraction state. Then,*

$$\text{inv}(MT, MA) \wedge \text{flushed-p}(MA) \rightarrow \text{MT-all-retired-p}(MT)\ .$$

When all instructions are retired, the MA is not speculatively executing any instructions. This is established in the following lemma:

Lemma 4 *Suppose MT is a MAETT. Then,*

$$\text{inv}(MT, MA) \wedge \text{MT-all-retired-p}(MT) \rightarrow \neg\text{MT-specultv-p}(MT) \ .$$

The predicate MT-self-modify-p(MT) holds iff any modified instructions are committed or currently being executed. It is possible that some modified instructions are executed speculatively, and the predicate MT-self-modify-p(MT) is true. On the other hand, MT-CMI-p(MT) holds only when some modified instruction have been completely executed and committed. It is easy to show the following lemma:

Lemma 5 *Suppose MT is a MAETT. Then,*

$$\text{MT-CMI-p}(MT) \rightarrow \text{MT-self-modify-p}(MT) \ .$$

Conversely, if all instructions recorded in MT are retired, and no modified instructions have been committed, then no instructions recorded in MT are modified.

Lemma 6 *Suppose MT is a MAETT. Then,*

$$\text{MT-all-retired-p}(MT) \wedge \neg\text{MT-CMI-p}(MT) \rightarrow \neg\text{MT-self-modify-p}(MT) \ .$$

The last lemma about self-modifying code relates the self-modification in the ISA model to the predicate MT-CMI-p(MT). In Figure 9.1, if the ISA execution from ISA_0 to ISA_m does not execute self-modifying code, no instructions recorded in the final MAETT MT_n are modified. Thus, MT_n does not satisfy MT-CMI-p(MT_n). This is formally proven in the following lemma.

Lemma 7 *Suppose MT satisfies* MAETT-p(MT). *Then,*

$$\neg\text{ISA-self-modify-p}(MT.\text{init-ISA}, \text{MT-exintr-lst}(MT), \text{MT-len}(MT))$$
$$\rightarrow \neg\text{MT-CMI-p}(MT) \ .$$

So far we have seen lemmas about speculative execution and self-modification of programs. Additionally, we need lemmas that relate the programmer visible states

in the final ISA state and the final MA state. In Subsection 8.1.15, we defined the predicates that relate the states of programmer visible components. For instance, the correct program counter value is tested with the predicate pc-match-p(MT, MA). From the definition of inv(MT, MA) and pc-match-p(MT, MA), we can easily show the following lemma.

Lemma 8

$$\begin{array}{ll} (& \text{inv}\,(MT,\ MA) \\ \wedge & (\neg\ \text{MT-specultv-p}\,(MT)) \\ \wedge & (\neg\ \text{MT-self-modify-p}\,(MT))) \\ \rightarrow & (\text{MT-pc}\,(MT) = MT.\text{pc}) \end{array}$$

As mentioned earlier, the function MT-pc(MT) defines the correct program counter value in state MA from its corresponding MAETT MT. This value is also the program counter value in the final ISA state. For instance in Figure 9.1, MT-pc(MT_n) is equal to the program counter value in ISA_m.

Lemma 9 *Suppose MT_n is a well-formed MAETT satisfying* weak-inv(MT_n). *Let*

$$\begin{array}{lcl} m & = & \text{MT-len}(MT_n) \\ ISA_m & = & \text{ISA-stepn}(MT_n.\text{init-ISA}, \text{MT-exintr-lst}(MT_n), m) \end{array}$$

Then,

$$\text{MT-pc}(MT_n) = ISA_m.\text{pc}\ .$$

This lemma can be proven from the definition of weak-inv(MT) and MT-pc(MT). By combining Lemma 8 and 9, we can show that the program counter value in MA_n is the same as in ISA_m in Figure 9.1.

Another invariant property RF-match-p(MT, MA) checks the correct register file state. From the definition of RF-match-p(MT, MA), we can show the following lemma.

Lemma 10

$$\text{inv}\,(MT,\ MA) \rightarrow (\text{MT-RF}\,(MT) = MA.\text{RF})$$

The function MT-RF(MT) defines the correct register file state in the corresponding MA state. The following lemma shows that the register file state defined by MT-RF(MT) is also the register file state for the final ISA state if all instructions recorded in MT are retired.

Lemma 11 *Suppose MT_n is a well-formed MAETT satisfying* weak-inv(MT_n). *Let*

$$\begin{aligned} m &= \text{MT-len}(MT_n) \\ ISA_m &= \text{ISA-stepn}(MT_n.\text{init-ISA}, \text{MT-exintr-lst}(MT_n), m) \end{aligned}$$

Then,

$$\text{MT-all-retired-p}(MT_n) \rightarrow \text{MT-RF}(MT_n) = ISA_m.\text{RF} \ .$$

From these lemmas, we can show the equivalence between the register file states in MA_n and ISA_m. Similarly, from the definition of SRF-match-p(MT, MA), we can show two lemmas about the state of the special register file.

Lemma 12

$$\text{inv}\,(MT,\ MA) \rightarrow (\text{MT-SRF}\,(MT) = (MA.\text{SRF}))$$

Lemma 13 *Suppose MT_n is a well-formed MAETT satisfying* weak-inv(MT_n). *Let*

$$\begin{aligned} m &= \text{MT-len}(MT_n) \\ ISA_m &= \text{ISA-stepn}(MT_n.\text{init-ISA}, \text{MT-exintr-lst}(MT_n), m) \end{aligned}$$

Then,

$$\text{MT-all-retired}(MT_n) \rightarrow \text{MT-SRF}(MT_n) = ISA_m.\text{SRF} \ .$$

For the memory state, the function MT-mem(MT) defines the correct memory state in the corresponding machine states.

Lemma 14

$$\text{inv}\,(MT,\ MA) \rightarrow (\text{MT-mem}\,(MT) = (MA.\text{mem}))$$

Lemma 15 *Suppose MT_n is a well-formed MAETT satisfying* weak-inv(MT_n). *Let*

$$\begin{aligned} m &= \text{MT-len}(MT_n) \\ ISA_m &= \text{ISA-stepn}(MT_n.\text{init-ISA}, \text{MT-exintr-lst}(MT_n), m) \end{aligned}$$

Then,

$$\text{MT-all-retired}(MT_n) \rightarrow \text{MT-mem}(MT_n) = ISA_m.\text{mem} \ .$$

From the lemmas and theorems presented above, we can prove the following correctness theorem.

Theorem 5 (Correctness Theorem) *Suppose MA_0, sig-list, and n are an MA state, a list of external signals to the MA, and a natural number, respectively. Let*

$$\begin{aligned} MT_0 &= \text{MT-init}(MA_0) \\ MA_n &= \text{MA-stepn}(MA_0, \textit{sig-list}, n) \\ MT_n &= \text{MT-stepn}(MT_0, MA_0, \textit{sig-list}, n) \\ \textit{intr-list} &= \text{MT-exintr-lst}(MT_n) \\ m &= \text{MT-len}(MT_n) \ . \end{aligned}$$

Then,

$$\begin{aligned} &\text{flushed-p}(MA_0) \wedge \text{flushed-p}(MA_n) \wedge \neg\text{ISA-self-modify-p}(\text{proj}(MA_0), \textit{intr-list}, m) \\ &\rightarrow \\ &\text{proj}(\text{MA-stepn}(MA_0, \textit{intr-list}, n)) = \text{ISA-stepn}(\text{proj}(MA_0), \textit{intr-list}, m) \end{aligned}$$

Proof: Assume the hypotheses of the theorem, flushed-p(MA_0), flushed-p(MA_n), and $\neg$ISA-self-modify-p(proj(MA_0), *intr-list*, m), and we prove the conclusion. Let

$$\begin{aligned} ISA_0 &= \text{proj}(MA_0) \\ ISA_m &= \text{ISA-stepn}(MT_0, \textit{intr-list}, m) \ . \end{aligned}$$

From the definition of MAETT, we can easily show $ISA_0 = MT_n$.init-ISA.

From Lemma 7 and the last hypothesis, we know that ¬MT-CMI-p(MT_n) is true. Using Theorem 4, we conclude that invariant inv(MT_n, MA_n) is true. From Lemma 3 and hypothesis flushed-p(MA_n), MT-all-retired-p(MT_n) holds. In other words, all instructions recorded in MT_n are retired. Thus, we can derive ¬MT-specultv-p(MT_n) from Lemma 4, and we can prove ¬MT-self-modify-p(MT_n) from Lemma 6.

Using Lemmas 8 and 9, we have

$$MA_n.\text{pc} = \text{MT-pc}(MT_n) = ISA_m.\text{pc} .$$

Since MT-all-retired-p(MT_n) holds, we obtain MT-RF(MT_n) = ISA_m.RF from Lemma 11. With Lemma 10,

$$MA_n.\text{RF} = \text{MT-RF}(MT_n) = ISA_m.\text{RF} .$$

Similarly, from Lemmas 12 and 13, we obtain

$$MA_n.\text{SRF} = \text{MT-SRF}(MT_n) = ISA_m.\text{SRF},$$

and from Lemmas 14 and 15,

$$MA_n.\text{mem} = \text{MT-mem}(MT_n) = ISA_m.\text{mem} .$$

From the four equalities above and the definition of proj(MA), we have

$$\text{proj}(MA_n) = ISA_m .$$

From the definition of ISA_m,

$$\text{proj}(MA_n) = \text{ISA-stepn}(\text{proj}(MA_0), \textit{sig-list}, m) \quad \square$$

The witness functions for our correctness criterion are:

$$W_N(MA, \textit{sig-list}, n) = \text{MT-len}(\text{MT-stepn}(\text{MT-init}(MA), MA, \textit{sig-list}, n))$$
$$W_{sig}(MA, \textit{sig-list}, n) = \text{MT-exintr-lst}(\text{MT-stepn}(\text{MT-init}(MA), MA, \textit{sig-list}, n)) .$$

In a nutshell, these witness functions construct the MAETT for the final flushed MA state, and count the number of instructions recorded in the MAETT and extract the values in the *exintr* field in INST representations of instructions. The function $W_N(MA, \textit{sig-list}, n)$ specifies the number of executed instructions during the MA execution and $W_{sig}(MA, \textit{sig-list}, n)$ specifies which instructions are actually interrupted by external signals.

Chapter 10

Verification Summary

Using the ACL2 theorem prover, we have completely verified our correctness criterion for the FM9801 design. In this chapter, we summarize the result of the verification project.

10.1 Cost Analysis

The verification cost is a serious practical concern for our technique. Since our verification was carried out solely by the ACL2 theorem prover, we had to manually write many lemmas to guide the prover. Table 10.1 shows the size of the ACL2 proof script files and the time to validate these files with a 200MHz PentiumPro system. The entire proof scripts consist of the FM9801 machine specification, the definition of the MAETT abstraction, the definition of invariant properties given in Table 8.1, a set of "shared lemmas", the proofs of the invariant properties, and the proof of the correctness criterion.

We wrote our ISA and MA specifications in one month, but the whole verification project took about 15 months. Notice that a large portion of the ACL2 script files is for proving invariant properties and basic lemmas. Since most of the

Table 10.1: ACL2 script size and CPU time for different verification phases.

Type of ACL2 Script	ACL2 Script Size	CPU Time to Certify
Specification of ISA and MA	140 KBytes	14 minutes
Definition of MAETT	55 KBytes	6 minutes
Definitions of Invariant Properties	89 KBytes	7 minutes
Proof of Basic Lemmas	481 KBytes	58 minutes
Proof of Invariant Properties	1034 KBytes	211 minutes
Proof of Correctness Criterion	37 KBytes	11 minutes

basic lemmas are, in fact, used for the proof of the invariant properties, we can safely say that the verification of invariant properties required most of our effort. This is not surprising because the verification of invariant properties is the core of our verification process. All design faults detected by applying formal verification techniques were found during this phase. Typically, a failed proof attempt returns the condition under which an invariant property is violated. We use it as a clue to identify a design fault in the microprocessor design.

Although the verification is labor intensive, our technique seems to scale well with the size of the verified design. In Table 10.2, we compare the size of our machine specification and verification scripts with two other proof efforts where we employed a similar approach. The ratio of the machine design and its verification script does not change much. We also note that the CPU time in Table 10.1 is relatively small, considering the size and complexity of the verified system. Our approach decomposes the verification of our correctness criterion into the verification of a number of invariant properties, which are further decomposed into sub-cases. This allows us to avoid possibly exponential case explosions. Typically, decomposed subgoals are small enough to be proven by the ACL2 theorem prover in a very short time. Of more than 6000 ACL2 theorems and commands in the FM9801 proof scripts, only 1.3 percent of them took more than a minute to be proven or executed.

From these results, we believe that our techniques do not suffer an expo-

Table 10.2: Sizes of ACL2 proof scripts for different machines. The small example machine is the three-stage pipelined machine discussed in Chapter 4.

Verified Machine	Machine Spec	Total Verification
Small Example Machine	13 KBytes	169 KBytes
5-stage Pipelined Design[SH97]	78 KBytes	757 KBytes
FM9801	140 KBytes	1909 KBytes

nential cost increase as the size of the verified machine grows. However, we need to reduce the cost of verification, especially that for the invariant properties. We may be able to apply more automated procedures for some invariant properties that involve a small number of hardware components. Our hope is that the best mix of a theorem prover environment with automated algorithmic verification procedures will reduce the overall cost of the verification.

10.2 Detected Design Faults

10.2.1 Overview

Not surprisingly, the initial design of the FM9801 contained many design flaws. First, we eliminated these design flaws with simulation techniques. Using the execution capability of the FM9801, we ran a few programs on the early design of the FM9801. We changed the external interrupt signals, memory responses, and branch prediction results by supplying different external input signals to the MA design, and tested the machine in various situations. This revealed most of the design faults in the early design of the FM9801. All design faults detected by simulation were fixed before we started applying formal verification techniques.

The formal verification phase started by defining the MAETT abstraction states of the FM9801 MA design, and specifying the invariant properties. We scrutinized the machine design during this phase and we found a few design faults. In

other words, carefully studying the machine design itself can reveal design faults. Although we do not consider these design flaws to be found by formal verification, we do consider this as a benefit of having formal specification.

After the invariant properties were specified, we started verifying the invariant properties and our correctness criterion. This verification phase detected 14 design faults which had not been detected by simulating and scrutinizing the design. Each time we detected a design flaw in the FM9801, we fixed the design and continued the formal verification process. Eventually, we completed the proof of our correctness criterion.

10.2.2 Details of Design Faults

In this subsection, we discuss the details of the design flaws detected by formal verification techniques. We explain them with a serial number, its classification, a brief description of the bug, a detailed description, and how we detected the design fault. We classify design faults into *bugs* and *glitches*. Bugs cause the processor to return incorrect results for some program execution. Glitches make the processor internally behave differently from the way the designer originally thought and violate some invariant properties. Glitches may or may not lead to an incorrect behavior visible to a programmer. We have found 12 bugs and 2 glitches. After fixing these 14 design faults, we successfully verified the entire MA design.

Design Fault Number: 1.

Classification: Glitch

Brief Description: Incorrect dispatching of instructions after exceptions were detected.

Description: This design fault caused the processor to incorrectly dispatch instructions with a fetch error exception or an illegal instruction exception. When instructions have raised a fetch error exception or an illegal instruction exception,

the instructions should not be dispatched to any execution unit, but they should go to the `'(complete)` stage directly. Instead, such instructions were dispatched to the integer unit regardless of the type of the instructions.

Due to this design fault, instructions with exceptions unnecessarily occupied the reservation station entries and consumed machine cycles in the integer unit, possibly causing a performance degradation. However, we could not find a single program execution which would have caused incorrect execution results. Consequently, this design fault was classified as a glitch.

How we found it: We have found the bug while verifying basic lemmas about stages of instructions. We tried to verify that instructions were dispatched to appropriate execution units, and we found any instruction could be dispatched to the integer unit if its exception status indicated that an exception had been detected.

Design Fault Number: 2.

Classification: Bug

Brief Description: The illegal instruction exception was not detected when the MTSR and MFSR instructions attempted to access non-existing special registers.

Description: When an MTSR instruction is executed with an *ra* field value other than 0 or 1, an illegal instruction exception should be detected according to the ISA specification, because the *ra* field should specify a special register *SR0* or *SR1*. However, this illegal instruction exception was not detected in the original design of the FM9801. The same bug occurred when an MFSR instruction was executed.

Design Fault Number: 3.

Classification: Bug

Brief Description: Incorrect values may overwrite correct operands stored in the reservation stations.

Description: Using Tomasulo's algorithm, the reservation station reads the value on the CDB when a tag match is found and uses it as an operand. For example,

the reservation stations for the multiply unit should read the value from the CDB to the field *val1* when the following conditions are met.

$$\begin{aligned} & \text{CDB-ready?}(MA) = 1 \\ \wedge \quad & \text{CDB-tag}(MA) = MA.\text{MU.RS0.src1} \\ \wedge \quad & MA.\text{MU.RS0.ready1?} \neq 1 \end{aligned}$$

In the original definition, the third condition was missing. Whenever the tag match occurred, the reservation station read the value from the CDB without checking the operand was already in the field *val1*. At first, we thought this was not a design fault because the tag should uniquely identify the instruction producing the operand. However, the tags were correct only when the corresponding operands were not ready. As a result, this bug could overwrite the correct operand value with an incorrect value.

How we found it: When we attempted the verification of the invariant property MT-inst-inv(MT, MA), the prover failed to prove the correctness of the intermediate values in the reservation stations. The condition under which the proof failed turned out to be the case where incorrect value may be overwritten.

Design Fault Number: 4.

Classification: Bug

Brief Description: The partial result is lost in the pipelined multiplier.

Description: The F9801 implements a three-stage pipelined multiplier with two internal latches *lch1* and *lch2*. If no instruction was in the latch *lch1*, and the instruction in the latch *lch2* stalled, the partial result in the *lch2* was lost.

How we found it: This bug was revealed when we tried to prove the correctness of the intermediate values at the latch *lch2*. A failed proof revealed the condition under which the partial result of a multiply instruction is lost.

Design Fault Number: 5.

Classification: Bug

Brief Description: A busy flag for the write buffer was not set properly.

Description: The write buffer has two entries: *wbuf0* and *wbuf1*. If a store instruction occupied the entry *wbuf0*, if the entry *wbuf1* was free, and if an instruction was issued to the write buffer at the same time the instruction at *wbuf0* was released, the issued instruction was lost because the busy flag for the entry *wbuf0* was not set.

How we found it: We found the bug when we tried to verify the correctness of the intermediate value at the stage `'(execute LSU wbuf1)`. The subgoal we failed to prove exhibited the condition that caused the improper behavior.

Design Fault Number: 6.

Classification: Bug

Brief Description: Load and store instructions were not issued correctly from a reservation station.

Description: The function issue-LSU-RS1?(*MA*) in the MA design should return 1 when an instruction is issued from reservation station 1. There was a typo in the definition of the function issue-LSU-RS1?(*MA*). As a result, the load instruction could be sent to a write buffer or a store instruction could be sent to a read buffer.

How we found it: When we tried to verify the correctness of the intermediate value at the stage `'(execute LSU wbuf1)`, we detected the case where we could not prove the correctness.

Design Fault Number: 7.

Classification: Bug

Brief Description: Load instructions returned incorrect results because of the bug in the load-bypassing logic.

Description: Load-bypassing allows load instructions to be executed without waiting for the completion of preceding store instructions when their memory access addresses differ. If an address match is found between load and store instructions, the

function address-match?(MA) should return 1. However, address-match?(MA) did not detect all address matches due to the bug. As a result, the load instruction was executed without waiting for the completion of a preceding store instruction with the same access address, and an incorrect value from the memory was returned as the loaded value. Instead, the correct implementation should use the load-forwarding technique and return the operand of the store instruction as the result of the load instruction.

How we found the bug: This bug was found while verifying the correctness of intermediate values for instructions at stage `'(execute LSU lch)`. When we tried to prove THEOREM LSU-forward-wbuf-INST-dest-val discussed in Subsection 8.2.4, we could not prove that the forwarded value, LSU-forward-wbuf(MA.LSU), was correct. We found that this was caused by a design fault in the definition of address-match?(MA).

It was difficult to find the program execution that revealed this bug to the programmer, because several load and store instructions had to be issued in a certain timing. The simplest example involved three instructions: a load instruction i_1, a store instruction i_2, and another load instruction i_3. In order to realize the bug, instruction i_1 had to stall in the read-buffer when instruction i_2 was issued, instruction i_3 had to be issued at the same time as the instruction i_1 read a value from the memory, and the memory access address of i_2 and i_3 must be equal. In this case, the instruction i_3 read the incorrect value of the memory before the completion of i_2.

Design Fault Number: 8.

Classification: Bug

Brief Description: The load-forwarding logic may forward the operand of a subsequent store instruction to a preceding load instruction.

Description: The read buffer of the load-store unit has the fields *wbuf0?* and

wbuf1?, which record the instruction order between load and store instructions. These fields might have been set incorrectly if a store instruction was released from the write buffer and simultaneously a load instruction was issued.

How we found it: We noticed this bug when we were scrutinizing the machine description to modify an invariant condition during the verification. In order to realize the bug, this bug also needed at least three load and store instructions issued and processed in a certain timing.

Design Fault Number: 9.

Classification: Bug

Brief Description: The store instruction in the write buffer may have been lost due to speculative execution.

Description: When a mispredicted branch instruction is committed, the subsequent instructions must be abandoned because they were speculatively executed. Due to the bug, this mechanism also flushed the content of the write buffers which might contain preceding store instructions. The correct implementation should check the instruction order and abandon only subsequent store instructions.

How we found it: When we tried to verify the intermediate values for an instruction at stage `'(commit wbuf0)`, we found that we could not prove that the busy flag was set properly.

Design Fault Number: 10.

Classification: Bug

Brief Description: The processor did not detect illegal instruction exceptions raised by executing an RFEH instruction in user mode.

Description: The RFEH instruction is a privileged instruction which should raise an exception when executed in user mode. In the original design, the processor failed to detect the exception and executed the instruction normally.

How we found it: While verifying invariant property SRF-match-p(MT, MA), we

needed to prove that the ISA and the MA modified the special register file in the same way. However, we found that the execution of RFEH instruction could change the special register file differently in the ISA and the MA models. It turned out that this happened when an RFEH instruction was executed in user mode.

Design Fault Number: 11.

Classification: Bug

Brief Description: Execution of an RFEH instruction updated the program counter incorrectly.

Description: This problem is similar to Design Fault 10. When the RFEH instruction was executed in user mode, the privileged instruction was executed without exceptions being detected. Fixing this problem introduced a new bug that set the program counter incorrectly when an RFEH instruction was executed in supervisor mode.

How we found it: After fixing Design Fault 10, we failed to verify the invariant property pc-match-p(MT, MA). Under the condition it failed, the RFEH instruction was found to set the program counter incorrectly.

Design Fault Number: 12.

Classification: Bug

Brief Description: Incorrect instructions were executed if branch predictions were performed more than once on a single branch instruction, and it was predicted differently.

Description: When a BR instruction is at the `'(IFU)` stage, the branch predictor predicts whether the conditional branch is taken. If the dispatch queue is full, this BR instruction stalls at the `'(IFU)` stage. As a result, branch prediction can be performed on a single BR instruction more than once. This could start fetching incorrect instructions in the following scenario:

1. The branch predictor predicts that a branch will be taken when the BR instruction at the `'(IFU)` stage is executed. This sets the program counter to the branch target address of the BR instruction. The BR instruction stalls and stays in the same stage.

2. The branch prediction is performed on the same BR instruction, and this time the branch is predicted not to be taken. This does not change the value of the program counter. As a result, the program counter continuously holds the branch target address.

3. The branch instruction advances to the next stage and the processor starts fetching instructions from the branch target address. However, the processor records that the branch was predicted not to be taken, because it is the result of the more recent branch prediction.

4. The branch execution unit decides that the branch instruction was not taken. The processor considers that it is executing correct subsequent instructions because its record shows that the branch is predicted not to be taken. In fact, the processor is executing instructions from the branch target address. Consequently, incorrect instructions from the branch target address will be completely executed.

How we found it: When we tried to verify the property pc-match-p(MT, MA), we found that the branch target address produced by the branch predictor was not always correct. It turns out that this occurs when a BR instruction stalled at the `'(IFU)` stage.

Design Fault Number: 13.

Classification: Bug

Brief Description: Register reference table for special registers do not record the correct tags.

Description: The logic of the register reference table was not working correctly. As a result, the tags stored in the register reference table might not identify the instructions that produce the newest value for the special registers. Consequently, MFSR and MTSR instructions might not be executed correctly.

How we found it: The invariant property consistent-SRF-tbl-p(MT, MA) could not be verified. Looking further into the problem, we found that the register reference table might not keep the correct tags for the special registers.

Design Fault Number: 14.

Classification: Glitch

Brief Description: The function commit-jmp?(MA), which should return 1 only when a branch instruction is committed, may also return 1 when an exception causing instruction is committed.

Description: The function commit-jmp?(MA) should return 1 when a mispredicted branch instruction is committed. Another function enter-excpt?(MA) returns 1 when the processor commits an instruction which has caused an exception. If they are asserted simultaneously, our MA design might not operate correctly, because we assumed in the design of the FM9801 that these functions are mutually exclusive. It turned out that commit-jmp?(MA) did not return the correct value for certain cases.

How we found it: When we tried to prove a lemma stating the mutual exclusion of commit-jmp?(MA) and enter-excpt?(MA), we found it unprovable. On the other hand, we could not find the program execution that simultaneously sets the values of the two functions to 1, and causes the machine to operate incorrectly. Consequently, it is classified as a glitch.

10.3 Summary

We found 12 bugs and 2 glitches, and some of them were difficult to find. For instance, Design Fault 1 may not affect the visible states of the FM9801, but it may degrade the performance by occupying resources and it may be classified as a performance bug. Design Fault 12 may be difficult to detect with simulation, because the bug is realized when multiple branch predictions return different results.

Each time we found a design fault and fixed the design of the FM9801 machine design, we reran ACL2 to check the entire proof. During this process, the robustness of the ACL2 proofs was found to be useful. There was a good chance that the same proof script worked for the slightly modified hardware design, because the proof specification does not depend on the subtle design specifics. Typically, when we change the design of the target machine, the ACL2 proofs fails only when it attempts to prove theorems directly related to the modified portion of the machine design. If we fix the proof of such theorems, the rest of the theorems are usually proven automatically.

Chapter 11

Conclusion

This dissertation has demonstrated that a complex pipelined microprocessor designs with advanced features can be formally verified. We have proposed a new microprocessor model called FM9801, and we have formally verified it using the ACL2 theorem prover. We consider that this is evidence that even complex pipelined microprocessor designs can be formally verified.

The main achievements of this dissertation are listed below.

- We have proposed correctness criteria for microprocessors with advanced pipelining techniques. Since pipelined microprocessors overlap the execution of instructions or sometimes interchange the order of execution, the states of pipelined microprocessors do not necessarily correspond to any sequential states which programmers have in mind. In fact, it is only in pipeline flushed states that we can see the direct correspondence between the sequential states and the pipeline states.

  For the correctness criteria for pipelined microprocessors, we proposed a commutative diagram that involves an arbitrary pipelined execution which starts and ends with pipeline flushed states. Compared with previously used commutative diagrams, our commutative diagrams can be applied to out-of-order

executions, speculative executions, and interrupts, which are implemented in the FM9801.

- We introduced an intermediate abstraction, MAETT, which helps to define invariants of pipelined machines. This abstraction records executed instructions in an ACL2 list in program order. This makes it easier to directly define properties about instructions.

- We have decomposed the pipeline verification by the following two steps. In the first step, we define and verify an invariant condition. In the next step, we prove our correctness criterion using the verified invariant as an assumption. Our invariant condition is a conjunction of many properties of the pipelined machine and its MAETT abstraction.

  This approach reduces the verification cost in two ways. First, our approach decomposes the verification problem over time. Instead of directly verifying the correctness criterion, we prove our correctness criterion from the separately verified invariant condition by induction. Our correctness criterion contains an arbitrary number of MA steps, but the invariant verification involves only one machine step. Thus the cost of verification does not suffer possibly exponential explosion with respect to the number of machine steps in the commutative diagram.

  Second, the verification of the invariant condition is decomposed spatially. Our invariant is defined as a conjunction of many properties. We prove these properties independently of each other. Typically, individual properties are related to only a small part of the microprocessor design. Thus, we can concentrate our computation and manual effort on the related microarchitectural components during the verification of individual properties.

- We studied self-modifying programs in the context of pipelined machine ver-

ification. Typically, pipelined microprocessors do not execute self-modifying programs as specified by a sequential execution model. In order to execute self-modifying code in pipelined microprocessors as specified by the ISA, the programmer usually has to run the modifying instructions first, synchronize the pipelined machine, and then execute the modified instructions. From our verification result, we can conclude that the FM9801 correctly executes self-modifying code when programmer explicitly synchronizes the program.

- We found a number of design faults in the FM9801. We fixed the FM9801 implementation each time a new fault was discovered, and continued the verification until it was completely finished. This demonstrates that our technique can be used to detect design errors in microprocessors.

Summarizing the results, we have successfully verified our FM9801 microprocessor. It is one of the most complex pipelined machines that have been completely verified, and contains a number of features that make the verification problem challenging.

According to the measurements obtained from our verification examples, our techniques seem to scale well with respect to the size of the machine design. We are optimistic that our technique can be applied to more complex microprocessor models. However, our technique currently requires a considerable amount of human interaction and expertise. The engineer who may want to use our technique must be knowledgeable not only about the microprocessor design, but also about the employed theorem proving system. In order to verify the microprocessor model, we needed 15 man-months of effort. Improving the efficiency of the verification is necessary to make our approach more acceptable.

Our technique is currently based solely on mechanical theorem proving. We have not yet integrated algorithmic approaches such as model checking into our techniques. Although it is our belief that algorithmic techniques cannot directly verify a large hardware design, such as the FM9801 microprocessor, it has the po-

tential to automate verification tasks required for our verification project. We have spent a significant amount of effort on establishing that the invariant is preserved by the FM9801 design. This often leads to tedious analyses of the local components and individual instructions. The verification of these local properties may be automated by using algorithmic approaches, which would improve the efficiency of our verification techniques.

Even though the FM9801 is not a toy example machine, it is far simpler than industrial microprocessor designs. We need more research to scale up our techniques to commercial microprocessors. At this moment, it may be more practical to apply our techniques to a portion of a commercial microprocessor. This may require us to reformulate the scheme of the verification. For example, our correctness criterion cannot be directly applied to a part of a microprocessor design.

At the conclusion of the dissertation, we now know that advanced microprocessor models can be formally verified. The verification cost is the largest problem at this moment, but we want to stress that the formally verified hardware design is invaluable from the perspective of security and mass production. We would like to see our work become the foundation of formal verification techniques used on future microprocessor designs.

Appendix A

A.1 Proof of Theorem 1

Theorem 1 proves our correctness criterion given in Criterion 1 assuming that Burch and Dill's flushing diagram holds. Let us note that flushing procedure applied to a flushed state returns the flushed state itself, i.e., flushed-p(MA) $\rightarrow$ flush(MA) = MA. Theorem 1 assumes MA-stepn(MA_0, n) is a flushed state. Thus,

$$\text{flush}(\text{MA-stepn}(MA_0, n)) = \text{MA-stepn}(MA_0, n).$$

Using this equality, Theorem 1 follows immediately from the following lemma:

Lemma 16 *Let MA_0 be a flushed state. Suppose for every i such that $0 \leq i < n$,*

$$\text{proj}(\text{flush}(\text{MA-step}(MA_i))) = \text{ISA-stepn}(\text{proj}(\text{flush}(MA_i)), k_i) \qquad \text{(A.1)}$$

for some k_i, where $MA_i = \text{MA-stepn}(MA_0, i)$. Then following equation must hold:

$$\text{proj}(\text{flush}(\text{MA-stepn}(MA_0, n))) = \text{ISA-stepn}(\text{proj}(MA_0), \sum_{i=0}^{n-1} k_i).$$

Proof: By induction on n. Base case is trivial as both sides equate to proj(MA_0). Induction case. our induction hypothesis is:

$$\text{proj}(\text{flush}(\text{MA-stepn}(MA_0, n-1)) = \text{ISA-stepn}(\text{proj}(MA_0), \sum_{i=0}^{n-2} k_i).$$

Then, the following equations hold:

$$
\begin{aligned}
&\text{proj}(\text{flush}(\text{MA-stepn}(MA_0, n))) \\
&= \text{proj}(\text{flush}(\text{MA-step}(MA_{n-1}))) && \{\text{Def.}\} \\
&= \text{ISA-stepn}(\text{proj}(\text{flush}(MA_{n-1})), k_{n-1}) && \{\text{A.1}\} \\
&= \text{ISA-stepn}(\text{ISA-stepn}(\text{proj}(MA_0), \textstyle\sum_{i=0}^{n-2} k_i), k_{n-1}) && \{\text{I.H.}\} \\
&= \text{ISA-stepn}(\text{proj}(MA_0), \textstyle\sum_{i=0}^{n-1} k_i) && \{\text{Def.}\}\ \square
\end{aligned}
$$

A.2 Theorem of Burch and Dill's Diagram Formation

In this section, we show Burch and Dill's diagram from a slightly modified version of our correctness criterion. We cannot directly show that our correctness criterion given in Definition 1 implies Burch and Dill's flushing diagram. The problem is that our correctness criterion does not say anything about pipeline flushing function flush(MA). In the proof of Theorem 6, we assume flushing procedure is a part of the MA execution that starts and ends with flushed pipeline states.

Pipeline flushing is an execution of the MA design without fetching new instructions. In the original paper by Burch and Dill, they define the pipelined machine that takes an external input, which controls instructions fetching. Only when this external signal is set to 1, the pipeline is allowed to fetch and execute new instructions. Pipeline flushing is performed by running the pipelined MA design for sufficiently many clock cycles with the external signal set to 0.

Let us consider a normal MA execution followed by pipelined flushing. We assume that we run the MA design from an initial flushed state MA_0 for n machine cycles to reach state MA_n, and then we flush the pipeline to reach flushed state MA'_n. We can represent MA'_n as flush(MA-stepn(MA_0, n)). Since the pipeline flushing procedure itself is an MA execution, the state transition from MA_0 to MA'_n can be considered as an MA execution starting and ending with a flushed state. Suppose N(MA_0, n) returns the number of instructions that are fetched during the

normal execution from MA_0 to MA_n. Further suppose the pipelined machine does not fetch instructions speculatively. $\mathrm{N}(MA_0, n)$ is the exact number of instructions that are completely executed during the execution from MA_0 to MA'_n, because no instructions are fetched during the flushing process.

Our correctness criterion is satisfied when the MA execution starting and ending with pipeline flushed states does have the same result as the ISA that executes the same number of instructions. This implies that the execution from MA_0 to MA'_n have the same result as the ISA that executes $\mathrm{N}(MA_0, n)$ instructions. This can be represented as:

$$\mathrm{proj}(\mathrm{flush}(\text{MA-stepn}(MA_0, n))) = \text{ISA-stepn}(\mathrm{proj}(MA_0), \mathrm{N}(MA_0, n)) \ .$$

Assuming this equation for an arbitrary flushed state MA_0 and any natural number n, we prove the flushing diagram for any MA state reachable from MA_0.

Theorem 6 *(Burch and Dill's Diagram Formation) Suppose there is a function* N *such that*

$$\mathrm{proj}(\mathrm{flush}(\text{MA-stepn}(MA_0, i))) = \text{ISA-stepn}(\mathrm{proj}(MA_0), \mathrm{N}(MA, i)) \tag{A.2}$$

for arbitrary flushed state MA_0 and any natural number i. Suppose $\mathrm{N}(MA, i)$ *is monotonic with respect to i, i.e., $i \leq j \rightarrow \mathrm{N}(MA, i) \leq \mathrm{N}(MA, j)$. Then the following equation representing flushing diagram holds:*

$$\begin{aligned} &\mathrm{proj}(\mathrm{flush}(\text{MA-step}(MA_n))) \\ &= \text{ISA-stepn}(\mathrm{proj}(\mathrm{flush}(MA_n)), \mathrm{N}(MA_n, n+1) - \mathrm{N}(MA_n, n)) \end{aligned} \tag{A.3}$$

where $MA_n = \text{MA-stepn}(MA_0, n)$.

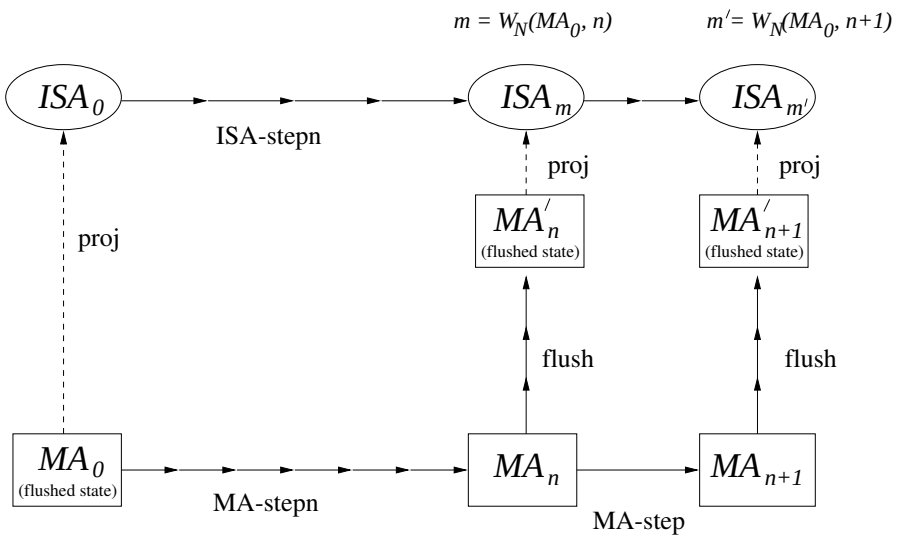

Figure A.1: Pictorial Proof of Theorem 6

Proof:

$$
\begin{aligned}
&\text{proj}(\text{flush}(\text{MA-step}(MA_n))) \\
&= \text{proj}(\text{flush}(\text{MA-stepn}(MA_0, n+1))) && \{\text{Def.}\} \\
&= \text{ISA-stepn}(\text{proj}(MA_0), \text{N}(MA_0, n+1)) && \{\text{(A.2), i = n+1}\} \\
&= \text{ISA-stepn}(\text{ISA-stepn}(\text{proj}(MA_0), \text{N}(MA_0, n)), \\
&\qquad \text{N}(MA_0, n+1) - \text{N}(MA_0, n)) && \{\text{Def. of ISA-stepn}\} \\
&= \text{ISA-stepn}(\text{proj}(\text{flush}(\text{MA-stepn}(MA_0, n))), \\
&\qquad \text{N}(MA_0, n+1) - \text{N}(MA_0, n)) && \{\text{(A.2), i = n}\} \\
&= \text{ISA-stepn}(\text{proj}(\text{flush}(MA_n)), \\
&\qquad \text{N}(MA_0, n+1) - \text{N}(MA_0, n)) && \{\text{Def.}\}\ \square
\end{aligned}
$$

Figure A.1 shows the relation between the states in the proof. Since equation A.2 may not hold for pipelined processors with speculative execution, this theorem is not usually applicable to such processors.

Appendix B

FM9801 State Definition

B.1 Definition of Words

The IHS macro, defbytetype, defines a word type. It defines the type predicate, the type coercion function, and related lemmas for each word type. Table B.1 shows the six word types we defined with the defbytetype macro.

The IHS macro, defword, can be used to define the field layout of a word type. For instance, the fields of the FM9801 16-bit instruction word, which are illustrated in Fig. 5.2, are defined with defword. Table B.2 shows the fields of an instruction word. Accessor functions take an instruction word, and return the field value. For

Type	Bit Width	Type Predicate	Type Coercion Function
word	16	word-p(x)	word(x)
addr	16	addr-p(x)	addr(x)
rname	4	rname-p(x)	rname(x)
immediate	8	immediate-p(x)	immediate(x)
opcd	4	opcd-p(x)	opcd(x)
cntlv	15	cntlv-p(x)	cntlv(x)

Table B.1: Words Defined with Defbytetype Macro

Field Name	Bit(s)	Accessor Function	Description
opcode	12-15	opcode(x)	Opcode
rc	8-11	rc(x)	Operand Register
ra	4-7	rb(x)	Operand Register
rb	0-3	rb(x)	Operand Register
im	0-7	im(x)	Immediate Value

Table B.2: Instruction Word Field Layout

Field Name	Bit(s)	Accessor	
exunit	10-14	exunit(x)	The Execution Unit for the instruction. Bit 14: No Execution unit. Bit 13: Branch unit. Bit 12: Load-store unit. Bit 11: Multiply unit. Bit 10: Integer unit.
operand	6-9	operand(x)	Specify the instruction format and operands. Bit 9: Format C, read a special register. Bit 8: Format C, read a general-purpose register. Bit 7: Format B. Bit 6: Format A.
br-predict?	5	br-predict(x)	The result of branch prediction.
ld-st?	4	ld-st(x)	Set to 1 for a store instruction, and 0 for a load.
wb?	3	wb(x)	The instruction modifies a register.
wb-sreg?	2	wb-sreg(x)	The modified register is a special register.
sync?	1	sync(x)	Synchronize the pipeline.
rfeh?	0	rfeh(x)	Set if the instruction is an RFEH.

Table B.3: Control Vector Field Layout for FM9801 Microarchitectural Design

instance, the opcode of an instruction word w is defined as opcode(w). Similarly, Table B.3 shows the layout of the control vector used in the microarchitectural design of the FM9801.

B.2 Definition of Register Files

Deflist RF **as List of** word **of Length** 16

Defstructure SRF {
    bitp su ; // Privilege Mode

word-p sr0 ; // Special Register 0
word-p sr1 ; // Special Register 1
}

Constructor Function:
SRF(*su*,*sr0*,*sr1*)

Type Predicate:
SRF-p(*SRF*)

Field Accessors:
.su .sr0 .sr1

Note: Access functions read-reg and read-sreg are defined differently, as the general-purpose register file is defined as a list of words, while the special register file is defined as a structure. The definitions of read-reg and read-sreg are as follows:

DEFINITION:
read-reg (num, RF) $\stackrel{def}{=}$ nth(num, RF)

DEFINITION:
read-sreg (id, SRF)
$\stackrel{def}{=}$
if id = `0` **then** SRF.sr0
elseif id = `1` **then** SRF.sr1
else `0`
fi

B.3 Definition of the ISA state

Defstructure ISA-state {
addr-p pc ; // Program Counter
rf-p rf ; // Register File
srf-p srf ; // Special Register File
mem-p mem ; // Memory
}

Constructor Function:
ISA-state(*pc*,*rf*,*srf*,*mem*)

Type Predicate:
ISA-state-p(*ISA*)

Field Accessors:
.pc .rf .srf .mem

B.4 Definition of the MA state

Defstructure MA-input {

bitp	exintr ;	// External Interrupt Signal
bitp	br-predict ;	// Branch Prediction Result
bitp	fetch ;	// Instruction Memory Response
bitp	data ;	// Data Memory Response

}

Constructor Function:
MA-input(*exintr*,*br-predict*,*fetch*,*data*)

Type Predicate:
MA-input-p(*orcl*)

Field Accessors:
.exintr .br-predict .fetch .data

Deflist MA-input-listp **as List of** MA-input-p

Defstructure IFU {

bitp	valid? ;	// Busy Flag
excpt-flags-p	excpt ;	// Exception Flags
addr-p	pc ;	// Program Counter Value
word-p	word ;	// Instruction Word

}

Constructor Function:
IFU(*valid?*,*excpt*,*pc*,*word*)

Type Predicate:
IFU-p(*IFU*)

Field Accessors:
.valid? .excpt .pc .word

Defstructure dispatch-entry {

bitp	valid? ;	// Busy Flag
excpt-flags-p	excpt ;	// Exception Flags
addr-p	pc ;	// Program Counter Value
cntlv-p	cntlv ;	// Control Vector
rname-p	rc ;	// Operand Register
rname-p	ra ;	// Operand Register
rname-p	rb ;	// Operand Register
immediate-p	im ;	// Immediate Value
addr-p	br-target ;	// Branch Target Address

}

Constructor Function:
dispatch-entry(*valid?*,*excpt*,*pc*,*cntlv*,*rc*,*ra*,*rb*,*im*,*br-target*)

Type Predicate:
dispatch-entry-p(*de*)

Field Accessors:
.valid? .excpt .pc .cntlv .rc .ra
.rb .im .br-target

Defstructure reg-ref {

bitp	wait? ;	// Register Write Pending
ROB-index-p	tag ;	// Last Modifier's Tag

}
Constructor Function:
reg-ref(*wait?*,*tag*)
Type Predicate:
reg-ref-p(*rr*)
Field Accessors:
.wait? .tag

Deflist reg-tbl-p **as List of** reg-ref-p **of Length** 16

Defstructure sreg-tbl {

reg-ref-p	sr0 ;	// Reference for SR0
reg-ref-p	sr1 ;	// Reference for SR1

}
Constructor Function:
sreg-tbl(*sr0*,*sr1*)
Type Predicate:
sreg-tbl-p(*srtbl*)
Field Accessors:
.sr0 .sr1

Defstructure DQ {

dispatch-entry-p	DE0 ;	// Dispatch Queue Entry 0
dispatch-entry-p	DE1 ;	// Dispatch Queue Entry 1
dispatch-entry-p	DE2 ;	// Dispatch Queue Entry 2
dispatch-entry-p	DE3 ;	// Dispatch Queue Entry 3
reg-tbl-p	reg-tbl ;	// Register Reference Table
sreg-tbl-p	sreg-tbl ;	// Special Register Reference Table

}
Constructor Function:
DQ(*DE0*,*DE1*,*DE2*,*DE3*,*reg-tbl*,*sreg-tbl*)
Type Predicate:
DQ-p(*DQ*)
Field Accessors:
.DE0 .DE1 .DE2 .DE3 .reg-tbl .sreg-tbl

Defstructure ROB-entry {

bitp	valid? ;	// Busy Flag
bitp	complete? ;	// Instruction Complete?
excpt-flags-p	excpt ;	// Exception Flags
bitp	wb? ;	// Write Back Instruction?
bitp	wb-sreg? ;	// Write to a Special Register?
bitp	sync? ;	// Synchronize Pipeline after Commit
bitp	branch? ;	// Branch Instruction
bitp	rfeh? ;	// RFEH Instruction
bitp	br-predict? ;	// Branch Prediction
bitp	br-actual? ;	// Actual Branch Direction
addr-p	pc ;	// Program Counter Value
word-p	val ;	// Instruction Result
rname-p	dest ;	// Destination Register

}

Constructor Function:

ROB-entry(*valid?*,*complete?*,*excpt*,*wb?*,*wb-sreg?*,*sync?*,*branch?*,*rfeh?*,*br-predict?*, *br-actual?*,*pc*,*val*,*dest*)

Type Predicate:

ROB-entry-p(*robe*)

Field Accessors:

.valid? .complete? .excpt .wb? .wb-sreg? .sync? .branch? .rfeh? .br-predict? .br-actual? .pc .val .dest

Deflist ROB-entries-p **as List of** ROB-entry-p **of Length** 8

Defstructure ROB {

bitp	flg ;	// Busy Flag
bitp	exintr? ;	// External Interrupt Pending?
ROB-index-p	head ;	// Head of the ROB
ROB-index-p	tail ;	// Tail of the ROB
ROB-entries-p	entries ;	// ROB Entry List

}

Constructor Function:

ROB(*flg*,*exintr?*,*head*,*tail*,*entries*)

Type Predicate:

ROB-p(*ROB*)

Field Accessors:

.flg .exintr? .head .tail .entries

Defstructure RS {

bitp	valid? ;	// Busy Flag
bitp	op ;	// Operation Type
ROB-index-p	tag ;	// Tag of the Instruction

```
    bitp            ready1? ;    // Operand 1 Ready?
    bitp            ready2? ;    // Operand 2 Ready?
    word-p          val1 ;       // Operand Value 1
    word-p          val2 ;       // Operand Value 2
    ROB-index-p     src1 ;       // Operand 1 Tag
    ROB-index-p     src2 ;       // Operand 2 Tag
    }
```

Constructor Function:

RS(*valid?*,*op*,*tag*,*ready1?*,*ready2?*,*val1*,*val2*,*src1*,*src2*)

Type Predicate:

RS-p(*RS*)

Field Accessors:

.valid? .op .tag .ready1? .ready2? .val1
.val2 .src1 .src2

Defstructure integer-unit {

```
    RS-p            RS0 ;        // Reservation Station 1
    RS-p            RS1 ;        // Reservation Station 2
    }
```

Constructor Function:

integer-unit(*RS0*,*RS1*)

Type Predicate:

integer-unit-p(*IU*)

Field Accessors:

.RS0 .RS1

Defstructure MU-latch1 {

```
    bitp            valid? ;     // Busy Flag
    ROB-index-p     tag ;        // Tag of the Instruction
    nil             data ;       // Abstract Data Value
    }
```

Constructor Function:

MU-latch1(*valid?*,*tag*,*data*)

Type Predicate:

MU-latch1-p(*lch*)

Field Accessors:

.valid? .tag .data

Defstructure MU-latch2 {

```
    bitp            valid? ;     // Busy Flag
    ROB-index-p     tag ;        // Tag of the Instruction
    nil             data ;       // Abstract Data Value
    }
```

Constructor Function:

MU-latch2(*valid?*,*tag*,*data*)

Type Predicate:

MU-latch2-p(*lch*)

Field Accessors:

.valid? .tag .data

Defstructure mult-unit {

RS-p	RS0 ;	// Reservation Station 0
RS-p	RS1 ;	// Reservation Station 1
MU-latch1-p	lch1 ;	// Latch 1
MU-latch2-p	lch2 ;	// Latch 2

}

Constructor Function:

mult-unit(*RS0*,*RS1*,*lch1*,*lch2*)

Type Predicate:

mult-unit-p(*MU*)

Field Accessors:

.RS0 .RS1 .lch1 .lch2

Defstructure LSU-RS {

bitp	valid? ;	// Busy Flag
bitp	op ;	// Operation Type
bitp	ld-st? ;	// Load or Store?
ROB-index-p	tag ;	// Tag of the Instruction
bitp	rdy3? ;	// Operand 3 Ready?
word-p	val3 ;	// Operand Value 3
ROB-index-p	src3 ;	// Operand Tag 3
bitp	rdy1? ;	// Operand 1 Ready
word-p	val1 ;	// Operand Value 1
ROB-index-p	src1 ;	// Operand Source 1
bitp	rdy2? ;	// Operand 2 Ready?
word-p	val2 ;	// Operand Value 2
ROB-index-p	src2 ;	// Operand Value 2

}

Constructor Function:

LSU-RS(*valid?*,*op*,*ld-st?*,*tag*,*rdy3?*,*val3*,*src3*,*rdy1?*,*val1*,
src1,*rdy2?*,*val2*,*src2*)

Type Predicate:

LSU-RS-p(*RS*)

Field Accessors:

.valid? .op .ld-st? .tag .rdy3? .val3
.src3 .rdy1? .val1 .src1 .rdy2? .val2
.src2

Defstructure read-buffer {

bitp	valid? ;	// Busy Flag
ROB-index-p	tag ;	// Tag of the Instruction
addr-p	addr ;	// Memory Access Address
bitp	wbuf0? ;	// Dependency with the Write in wbuf0
bitp	wbuf1? ;	// Dependency with the Write in wbuf1

}

Constructor Function:
read-buffer(*valid?*,*tag*,*addr*,*wbuf0?*,*wbuf1?*)

Type Predicate:
read-buffer-p(*rbuf*)

Field Accessors:
.valid? .tag .addr .wbuf0? .wbuf1?

Defstructure write-buffer {

bitp	valid? ;	// Busy Flag
bitp	complete? ;	// Memory Protection Check Done?
bitp	commit? ;	// Instruction Committed?
ROB-index-p	tag ;	// Tag of the Instruction
addr-p	addr ;	// Memory Access Address
word-p	val ;	// Write Value

}

Constructor Function:
write-buffer(*valid?*,*complete?*,*commit?*,*tag*,*addr*,*val*)

Type Predicate:
write-buffer-p(*wbuf*)

Field Accessors:
.valid? .complete? .commit? .tag .addr .val

Defstructure LSU-latch {

bitp	valid? ;	// Busy Flag
excpt-flags-p	excpt ;	// Exception Flags
ROB-index-p	tag ;	// Tag of the Instruction
word-p	val ;	// Result Value from Memory Load

}

Constructor Function:
LSU-latch(*valid?*,*excpt*,*tag*,*val*)

Type Predicate:
LSU-latch-p(*lch*)

Field Accessors:
.valid? .excpt .tag .val

Defstructure load-store-unit {

```
    bitp                RS1-head? ;  // Order of RS0 and RS1
    LSU-RS-p            RS0 ;        // Reservation Station 0
    LSU-RS-p            RS1 ;        // Reservation Station 1
    read-buffer-p       rbuf ;       // Read Buffer
    write-buffer-p      wbuf0 ;      // Write Buffer Entry 0
    write-buffer-p      wbuf1 ;      // Write Buffer Entry 1
    LSU-latch-p         lch ;        // Result Latch
    }
```

Constructor Function:

load-store-unit(*RS1-head?*,*RS0*,*RS1*,*rbuf*,*wbuf0*,*wbuf1*,*lch*)

Type Predicate:

load-store-unit-p(*LSU*)

Field Accessors:

.RS1-head? .RS0 .RS1 .rbuf .wbuf0 .wbuf1 .lch

Defstructure BU-RS {

```
    bitp                valid? ;     // Busy Flag
    ROB-index-p         tag ;        // Tag of the Instruction
    bitp                ready? ;     // Operand Ready
    word-p              val ;        // Operand Value
    ROB-index-p         src ;        // Operand Tag
    }
```

Constructor Function:

BU-RS(*valid?*,*tag*,*ready?*,*val*,*src*)

Type Predicate:

BU-RS-p(*RS*)

Field Accessors:

.valid? .tag .ready? .val .src

Defstructure branch-unit {

```
    BU-RS-p             RS0 ;        // Reservation Station 0
    BU-RS-p             RS1 ;        // Reservation Station 1
    }
```

Constructor Function:

branch-unit(*RS0*,*RS1*)

Type Predicate:

branch-unit-p(*BU*)

Field Accessors:

.RS0 .RS1

Defstructure MA-state {

```
    addr-p              pc ;         // Program Counter
    RF-p                RF ;         // General-Purpose Register File
```

```
    SRF-p              SRF ;        // Special Register File
    IFU-p              IFU ;        // Instruction Fetch Unit
    DQ-p               DQ ;         // Dispatch Queue
    ROB-p              ROB ;        // Re-order Buffer
    integer-unit-p     IU ;         // Integer Unit
    mult-unit-p        MU ;         // Multiply Unit
    branch-unit-p      BU ;         // Branch Unit
    load-store-unit-p  LSU ;        // Load Store Unit
    mem-p              mem ;        // Memory
    }
```

Constructor Function:

MA-state(*pc*,*RF*,*SRF*,*IFU*,*DQ*,*ROB*,*IU*,*MU*,*BU*, *LSU*,*mem*)

Type Predicate:

MA-state-p(*MA*)

Field Accessors:

.pc .RF .SRF .IFU .DQ .ROB .IU .MU .BU .LSU .mem

B.5 Definition of the MAETT state

Defstructure INST {

```
    naturalp           ID ;             // Identification Number
    bitp               modified? ;      // Modified by Self-Modifying Code?
    bitp               first-modified? ; //
    bitp               speculative? ;   // Speculatively Executed?
    bitp               br-predict? ;    // Branch Prediction Result
    bitp               exintr? ;        // Externally Interrupted
    word-p             word ;           // Instruction Word
    stage-p            stg ;            // Current Stage
    ROB-index-p        tag ;            // Tag used in Tomasulo's Algorithm
    ISA-state-p        pre-ISA ;        // Pre-ISA state
    ISA-state-p        post-ISA ;       // Post-ISA state
    }
```

Constructor Function:

INST(*ID*,*modified?*,*first-modified?*,*speculative?*,*br-predict?*,*exintr?*,*word*,*stg*,*tag*, *pre-ISA*,*post-ISA*)

Type Predicate:

INST-p(*i*)

Field Accessors:

.ID .modified? .first-modified? .speculative? .br-predict? .exintr? .word .stg .tag .pre-ISA .post-ISA

Deflist INST-listp **as List of** INST-p

Defstructure MAETT {

ISA-state-p	init-ISA ;	//
naturalp	new-ID ;	// ID for Newly Fetched Instruction
naturalp	DQ-len ;	// Number of Instructions in Dispatch Queue
naturalp	WB-len ;	// Number of Instructions in Write Buffer
bitp	ROB-flg ;	// ROB-head is less than or equal to ROB-tail.
ROB-index-p	ROB-head ;	// Head of Reorder Buffer
ROB-index-p	ROB-tail ;	// Tail of Reorder Buffer
INST-listp	trace ;	// List of Executed Instructions

}

Constructor Function:

MAETT(*init-ISA*,*new-ID*,*DQ-len*,*WB-len*,*ROB-flg*,*ROB-head*,*ROB-tail*,*trace*)

Type Predicate:

MAETT-p(*MT*)

Field Accessors:

.init-ISA .new-ID .DQ-len .WB-len .ROB-flg .ROB-head .ROB-tail .trace

Appendix C

List of INST Functions

In this Appendix, we list the functions that calculate various values for instructions. Some of these functions are introduced in Subsection 7.3.2, and used in the following sections. The complete definitions of these functions are given in Appendix D.

Function Name	Description
INST-word(i)	Instruction word.
INST-pc(i)	Program counter value, or the address of instruction word.
INST-RF(i)	Register file before executing i.
INST-SRF(i)	Special register file before executing i.
INST-mem(i)	Memory before executing i.
INST-su(i)	Supervisor/User mode.
INST-opcode(i)	Opcode.
INST-ra(i)	RA operand register.
INST-rb(i)	RB operand register.
INST-rc(i)	RC operand register.
INST-im(i)	Immediate value.
INST-fetch-error?(i)	Causes a fetch error if 1.
INST-decode-error?(i)	Causes an illegal instruction if 1.
INST-load-error?(i)	Causes a read memory exception if 1.
INST-store-error?(i)	Causes a write memory exception if 1.
INST-data-access-error?(i)	Causes a data access error exception if 1.
INST-excpt?(i)	Causes an exception of any kind if 1.

Function Name	Description
INST-cntlv(i)	Control vector.
INST-load-addr(i)	Memory load address if load instruction.
INST-store-addr(i)	Memory store address if store instruction.
INST-src-val1(i)	First source operand value.
INST-src-val2(i)	Second source operand value.
INST-src-val3(i)	Third source operand value.
INST-ADD-dest-val(i)	Result (destination value) of ADD instruction.
INST-MULT-dest-val(i)	Result of MUL instruction.
INST-LD-dest-val(i)	Result of LD instruction.
INST-LD-im-dest-val(i)	Result of LDI instruction.
INST-MFSR-dest-val(i)	Result of MFSR instruction.
INST-MTSR-dest-val(i)	Result of MTSR instruction.
INST-writeback-p(i)	Instruction write back its result to a register.
INST-dest-val(i)	Result of an instruction
INST-dest-reg(i)	Destination register
INST-IU?(i)	Executed in the integer unit if 1.
INST-MU?(i)	Executed in the multiply unit if 1.
INST-LSU?(i)	Executed in the load store unit if 1.
INST-BU?(i)	Executed in the branch unit if 1.
INST-no-unit(i)	Not executed in any unit.
INST-ld-st(i)	Control vector flag ld-st.
INST-store(i)	Memory store instruction.
INST-load(i)	Memory load instruction.
INST-wb(i)	Control vector flag wb.
INST-wb-sreg(i)	Control vector flag wb-sreg.
INST-sync(i)	Control vector flag sync.
INST-rfeh(i)	Control vector flag rfeh.
INST-branch-dest(i)	Branch target address.
INST-IU-op(i)	Operand type for instructions executed in IU.
INST-LSU-op(i)	Operand type for instructions executed in LSU.

Function Name	Description
INST-context-sync?(i)	Context switching instruction.
INST-branch-taken?(i)	Branch is taken.
INST-wrong-branch?(i)	Branch is mispredicted.
INST-start-specultv?(i)	Instruction starts speculative execution.
INST-fetch-error-detected-p(i)	A fetch error is detected.
INST-decode-error-detected-p(i)	A decode error is detected.
INST-load-accs-error-detected-p(i)	A load access error is detected.
INST-store-accs-error-detected-p(i)	A store access error is detected.
INST-data-accs-error-detected-p(i)	A data access error is detected.
INST-excpt-detected-p(i)	An exception is detected by the processor.
INST-excpt-flags(i)	An exception flag is raised.

Appendix D

ACL2 Books for the FM9801 Verification

This Appendix is the ACL2 specifications of the FM9801 microprocessor and its verification scripts. The hierarchy of the entire proof scripts is shown in Figure D.1, except that several basic files are omitted from the figure. The excluded files are given in Section D.1. The hierarchy of the proof scripts is divided into the machine definition layer, the intermediate abstraction layer, the invariant definition layer, the shared lemma layer, the invariant proof layer, and the correctness proof layer. The machine definition layer gives the the FM9801 definitions in three files, `basic-def.lisp`, `ISA-def.lisp`, and `MA-def.lisp`. The definition of the MAETT intermediate abstraction is given in `MAETT-def.lisp` in the intermediate abstraction layer. On top of this abstraction, `invariants-def.lisp` defines the invariant condition of the FM9801. The shared lemma layer contains general lemmas about the FM9801 and the MAETT abstraction which are used in the proofs in the invariant proof layer and the correctness proof layer. The invariant proof layer collects all the scripts proving the invariant condition. The FM9801 correctness criterion is proven in correct.lisp. At the moment this dissertation is written, the entire proof script is

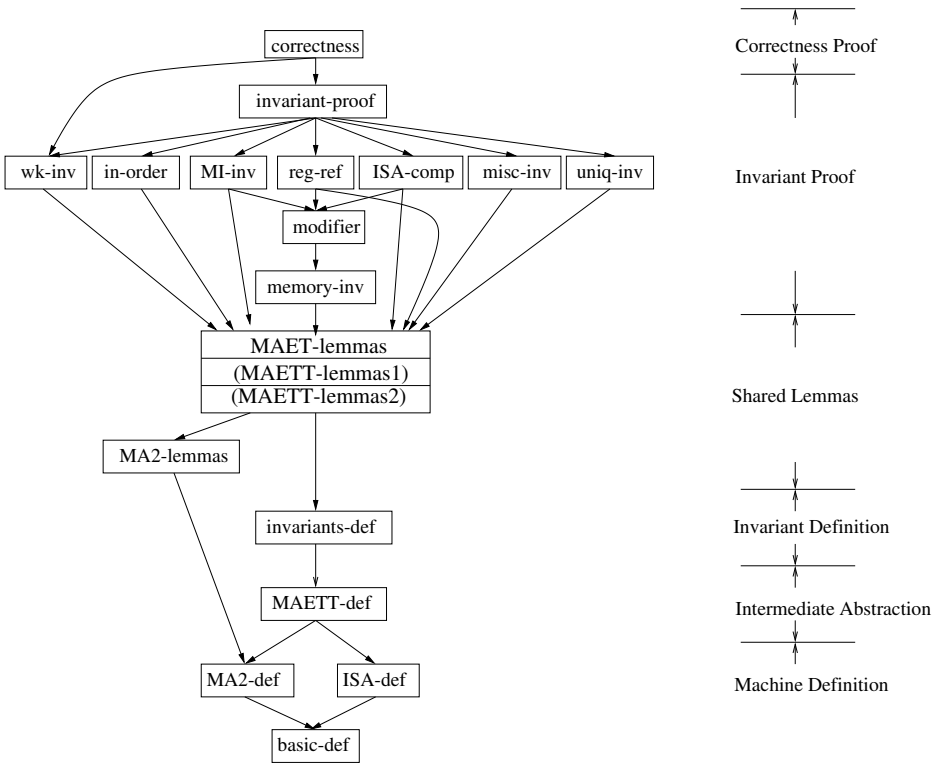

Figure D.1: Hierarchy of the FM9801 proof scripts. Suffix ".lisp" is omitted in this figure.

available from the website: `http://www.cs.utexas.edu/users/sawada/FM9801`.

D.1 Basic Books for FM9801 Verification

This section contains a various files used to prove basic theorems which are not directly related to the FM9801.

D.1.1 absolute-path.lisp

This defines a macro to load ACL2 public libraries.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ihs-def.lisp:
; Author  Jun Sawada, University of Texas at Austin
;
; A macro to define ACL2 public libraries.
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(in-package "ACL2")

(defmacro acl2-book-dir ()
  "/p/lib/acl2/books")

(defmacro include-acl2-book
    (book-name
     &key (load-compiled-file ':warn)
     doc)
  (declare (xargs :guard
                  (member load-compiled-file
                          '(t nil :warn :try))))
  `(include-book
    ,(concatenate 'string
                  (acl2-book-dir)
                  "/" book-name)
    :load-compiled-file ,load-compiled-file
    :doc ,doc))
```

D.1.2 IHS.lisp

This file loads the ACL2 V2-3 IHS library and set the appropriate ACL2 theory environment.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ihs-def.lisp:
; Author  Jun Sawada, University of Texas at Austin
;
; This file loads the IHS library and adjust the ACL2 theory.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(in-package "ACL2")

(include-book "absolute-path")
(deflabel before-loading-ihs)
(include-acl2-book "ihs/ihs-definitions")
(include-acl2-book "ihs/ihs-lemmas")
(deflabel after-loading-ihs)

(deftheory default-IHS-incompatibles
    '(oddp evenp floor mod rem truncate expt
      loghead logtail logior lognot logand logand
      logeqv logorc1 logorc2 logandc1 logandc2 logcount
      lognand lognor logxor))

(in-theory
 (set-difference-theories (current-theory :here)
          (set-difference-theories  (universal-theory 'after-loading-IHS)
                                    (universal-theory 'before-loading-IHS))))

(in-theory (disable default-ihs-incompatibles))
(in-theory
 (enable
  ihs-math                              ; From "math-lemmas"
  quotient-remainder-rules              ; From "quotient-remainder-lemmas"
  logops-lemmas-theory))                ; From "logops-lemmas"

(in-theory (disable (force)))
```

```
(in-theory (disable
            (:generalize MOD-X-Y-=-X+Y)
            (:generalize MOD-X-Y-=-X)
            (:generalize MOD-=-0)
            (:generalize FLOOR-TYPE-4)
            (:generalize FLOOR-TYPE-3)
            (:generalize FLOOR-TYPE-2)
            (:generalize FLOOR-TYPE-1)
            (:generalize FLOOR-BOUNDS)
            (:generalize FLOOR-=-X/Y)))


(defun defword-tuple-typecheck-thms (tuple)
  (let ((tuple-thm-name
         (pack-intern (first tuple) "DEFWORD-TUPLE-" (first tuple))))
    `(local (defthm ,tuple-thm-name
              (and
               (integerp ,(second tuple))
               (> ,(second tuple) 0)
               (integerp ,(third tuple))
               (>= ,(third tuple) 0)
               (implies ,(fourth tuple) (stringp ,(fourth tuple))))
              :rule-classes nil))))


(defun defword-struct-typecheck-thms (struct)
  (if (endp struct) nil
      (cons (defword-tuple-typecheck-thms (car struct))
            (defword-struct-typecheck-thms (cdr struct)))))

(defun type-check-thms
    (name struct conc-name set-conc-name keyword-updater doc)
  (append
   (list
    `(local (defthm defword-symbolp-name (symbolp ',name)
              :rule-classes nil
              :doc "Defword name should be a symbol."))
    `(local (defthm defword-symbolp-conc-name (symbolp ',conc-name)
              :rule-classes nil
              :doc "Defword conc-name should be a symbol."))
    `(local (defthm defword-symbolp-set-conc-name (symbolp ',set-conc-name)
              :rule-classes nil
              :doc "Defword set-conc-name should be a symbol."))
    `(local
      (defthm defword-symbolp-keyword-updater (symbolp ',keyword-updater)
        :rule-classes nil
        :doc "Defword keyword-updater should be a symbol."))
    `(local (defthm defword-stringp-doc
              (implies ',doc (stringp ',doc))
              :rule-classes nil
              :doc "Defword doc should be a string.")))
   (defword-struct-typecheck-thms struct)))


;; This version of defword postpones the type checking of arguments until
;; macro expansion is completed.  Because the original defword checks
;; types of arguments before expanding macro, some defword expressions
;; are rejected even if it can be considered as a legitimate defword
;; expression. For instance,
;; (defword new-word ((field1 *field1-pos* *field1-width*)
;;                    (field2 *field2-pos* *field2-width*)))
```

```
;; is rejected because *field1-pos* and other constants are not
;; integers.
(defmacro defword* (name struct &key conc-name set-conc-name keyword-updater
                         doc)
  (cond
   ((not (weak-defword-guards struct)))
   (t
    (let*
      ((conc-name (if conc-name
                      conc-name
                    (pack-intern name name "-")))
       (set-conc-name (if set-conc-name
                          set-conc-name
                        (pack-intern name "SET-" name "-")))
       (keyword-updater (if keyword-updater
                            keyword-updater
                          (pack-intern name "UPDATE-" name)))
       (type-check-thms
        (type-check-thms name struct conc-name set-conc-name
                         keyword-updater doc))
       (accessor-definitions
        (defword-accessor-definitions 'RDB name conc-name struct))
       (updater-definitions
        (defword-updater-definitions 'WRB name set-conc-name struct))
       (field-names (strip-cars struct)))

      `(ENCAPSULATE ()                  ;Only to make macro expansion pretty.
         (DEFLABEL ,name ,@(if doc `(:DOC ,doc) nil))
         ,@type-check-thms
         ,@accessor-definitions
         ,@updater-definitions
         ,(defword-keyword-updater
            name keyword-updater set-conc-name field-names))))))
```

D.1.3 trivia.lisp

This file proves basic ACL2 lemmas which are used in the proof.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Trivia.lisp
; Author  Jun Sawada, University of Texas at Austin
;
; This file proves basic lemmas about ACL2 functions.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;'
(in-package "ACL2")

(include-book "absolute-path")
(include-acl2-book "data-structures/array1")
(include-acl2-book "arithmetic/top")

(deflabel begin-trivia)

(defmacro quoted-constant-p (x)
  `(and (consp ,x) (equal (car ,x) 'quote)))

(defthm append-nil
    (implies (true-listp x)
             (equal (append x nil) x))
  :hints (("Goal" :in-theory (enable binary-append))))
```

```
(defun natural-induction (i)
  (if (zp i) t
      (natural-induction (1- i))))

(defthm car-nthcdr
    (equal (car (nthcdr idx lst)) (nth idx lst)))

(defthm cdr-nthcdr
    (implies (and (integerp idx) (<= 0 idx))
             (equal (cdr (nthcdr idx lst)) (nthcdr (+ 1 idx) lst))))

(in-theory (disable length))

(in-theory (disable array1p compress1 default dimensions header
                    aref1 aset1 maximum-length))

(defthm array1p-module-type
    (implies
     (array1p name l)
     (and (symbolp name)
          (alistp l)
          (consp (header name l))
          (consp (dimensions name l))
          (true-listp (dimensions name l))
          (integerp (car (dimensions name l)))
          (<= 0 (car (dimensions name l)))
          (integerp (maximum-length name l))
          (<= 0 (maximum-length name l))))
  :hints (("Goal" :in-theory (enable array1p header default dimensions
                                     maximum-length)))
  :rule-classes
  ((:type-prescription :corollary
                       (implies
                        (array1p name l)
                        (and (consp (header name l)))))
   (:type-prescription :corollary
                       (implies
                        (array1p name l)
                        (and (consp (dimensions name l))
                             (true-listp (dimensions name l)))))
   (:type-prescription :corollary
                       (implies
                        (array1p name l)
                        (and (integerp (car (dimensions name l)))
                             (<= 0 (car (dimensions name l))))))
   (:type-prescription :corollary
                       (implies
                        (array1p name l)
                        (and (integerp (maximum-length name l))
                             (<= 0 (maximum-length name l)))))))

(local
(defthm compress11-empty-array
    (implies (and (integerp n) (>= n 0))
             (equal (compress11 name (list (cons :header info)) n dim default)
                    nil))
  :hints (("Goal" :In-theory (enable compress11)))))

(defthm compress1-empty-array
    (equal (compress1 tag (list (cons :header x)))
           (list (cons :header x)))
  :hints (("Goal" :in-theory (enable compress1 compress11
```

```
                                    header default))))

(defthm array1p-cons-with-dimensions
    (implies (and (array1p tag array)
                  (integerp index)
                  (>= index 0)
                  (> (car (dimensions tag array)) index))
             (array1p tag (cons (cons index val) array)))
  :hints (("goal" :in-theory (enable dimensions header))))
```

D.1.4 define-u-package.lisp

This file defines a package named "U".

```
;; Define the U package.

(in-package "ACL2")

(defpkg "U" (union-eq *acl2-exports*
                      *common-lisp-symbols-from-main-lisp-package*))
```

D.1.5 utils.lisp

This file provides ACL2 utility functions for ACL2 theory definition, reloading the ACL2 file, and computational hints.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; utils.lisp
; Author  Jun Sawada, University of Texas at Austin
;
; ACL2 utility functions and macros.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(in-package "ACL2")

(include-book "absolute-path")
(include-acl2-book "data-structures/utilities")

(u::import-as-macros
 U::A-UNIQUE-SYMBOL-IN-THE-U-PACKAGE
 defloop)

(defloop non-rec-functions (runes world)
  (for ((rune in runes))
       (unless (fgetprop (cadr rune) 'induction-machine nil world)
         (collect rune))))

(defloop rec-functions (runes world)
  (for ((rune in runes))
       (when (fgetprop (cadr rune) 'induction-machine nil world)
         (collect rune))))

(defmacro ww (form)
  "With (W state) ..."
  `(LET ((WORLD (W STATE))) ,form))

;Example
;(ww (non-rec-functions (definition-theory (current-theory :here)) world))
```

```
;Example
; (deftheory test (non-rec-functions (theory 'table-def) world))
;
; Macro ld-up-to fast loads ACL2 events in a file to a specified point.
;  (ld-up-to "<filename>" <event_name>  {:speed <speed> })
; This command loads ACL2 file <filename> to the point where
; <event_name> is first defined.

; Since ld-up-to skips proofs of the theorems in a loaded file by
; default, a user can quickly reach the state where he or she wants
; to work in.  <even_name> can be any symbol newly defined by the
; event at the desired breaking point.  For instance, a label
; defined by deflabel can be used as well.  Keyword :all can be
; specified when the user want to load the whole file.

;
; The user can specify loading speed using keyword :speed
;  :speed 0  Does not skip proofs. The slowest way to load a file.
;  :speed 1  Skips proofs, but performs other checks on theorems.
;            Ld-up-to loads files at this speed by default.
;  :speed 2  Skips proofs and other checks on theorems.  Warning
;            will not be printed out.  It also skips events local to
;            the ACL2 file.

; Example
;  (ld-up-to "invariants-def.lisp" invariants)
;   This command loads ACL2 file invariant-def.lisp until it finds the
;   definition of function invariants.
;
;  (ld-up-to "invariants-def.lisp" :all :speed 2)
;   this command load the all events in invariants-def.lisp except
;   events local to the ACL2 book.
(defmacro ld-up-to (file event &key (speed '1))
  (let ((skip-proofs-flag (if (equal speed 0) nil
                            (if (equal speed 1) t ''include-book))))
    `(ld ,file :ld-pre-eval-filter ',event
      :ld-skip-proofsp ,skip-proofs-flag)))

; Macro refresh rolls back ACL2 history and reloads ACL2 events fast.
;
; (refresh <filename> {:back-to <event1>} {:up-to <event2>} {:speed <speed>})
;
; Refresh first undoes events back to the point where <event1> was
; defined, then it loads ACL2 file <filename> up to the point where
; <event2> is newly defined.  When previously executed ACL2 events
; like definitions and theorems are modified, the current world of ACL2
; is corrupted and a user may want to restart the ACL2 from scratch.
; However, restarting ACL2 takes a long time especially if many books
; are loaded.  Refresh helps to reload modified files and get the
; newest state of ACL2 world quickly.  It only undoes to the point
; where symbol <event1> is defined, and then loads file <filename>,
; skipping proofs by default.  Since refresh loads ACL2 books included
; by <filename>, usually the user only has to specify the top ACL2 book.
; <event1> can be a symbol defined in an included book, and refresh
; undoes the include-book of the included book.  If :back-to
; keyword is not supplied, refresh undoes all user defined events.  If
; :up-to keyword is not given, it load all events in <filename>.  So
; (refresh <filename>) is equivalent to (refresh <filename> :back-to 1
; :up-to :all).  <event1> can be a number designating an event.
;
; The user can also specify loading speed using keyword :speed
```

```
;  :speed 0  Does not skip proofs. The slowest way to load a file.
;  :speed 1  Skips proofs, but performs other checks on theorems.
;            Ld-up-to loads files at this speed by default.
;  :speed 2  Skips proofs and other checks on theorems.  Warning
;            will not be printed out.  It also skips events local to
;            the ACL2 file.

(defmacro refresh (file &key (back-to '1) (up-to ':all) (speed '1))
  (let ((skip-proofs-flag (if (equal speed 0) nil
                            (if (equal speed 1) t ''include-book))))
    `(ld '((ubt! ',back-to) (ld ,file :ld-pre-eval-filter ',up-to
                              :ld-skip-proofsp ,skip-proofs-flag)))))

;;;;;;;
; A computed hint
;;;;;;
(defmacro use-hint-always (hint)
  `',hint)

; When-found is a macro to supply a computational hint.  When term is
; found in the goal clause, hint is invoked.  An example usage follows:
;  :hints ((when-found (FETCHED-INST MT (MT-FINAL-ISA MT)
;                                    (MT-IN-SPECULTV? MT))
;                      (:cases ((b1p (MT-IN-SPECULTV? MT))))))
;
(defmacro when-found (term hint)
  `(and (occur-lst ',term clause) ',hint))

(defun multiple-occur-check (terms)
  (if (endp terms)
      nil
      (if (endp (cdr terms))
          `(occur-lst ',(car terms) clause)
          `(and (occur-lst ',(car terms) clause)
            ,(multiple-occur-check (cdr terms))))))

(defmacro when-found-multiple (terms hint)
  `(and ,(multiple-occur-check terms) ',hint))

(defmacro show-hint (hint &optional marker)
  (cond
   ((and (consp hint)
         (stringp (car hint)))
    hint)
   (t
    `(let ((marker ,marker)
           (ans ,(if (symbolp hint)
                     `(,hint id clause world)
                   hint)))
       (if ans
           (prog2$
            (cw "~%***** Computed Hint~#0~[~/ (from hint ~x1)~]~%~x2~%~%"
                (if (null marker) 0 1)
                marker
                (cons (string-for-tilde-@-clause-id-phrase id)
                      ans))
            ans)
         nil)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Pattern match functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
; do not check if x is a cons.
(defun fmeta-varp (x)
  (and (equal (car x) '@) (symbolp (cadr x))))

(defun fmeta-var-name (x) (cadr x))

(defmacro mv2-or (first second)
  `(mv-let (flg val) ,first
    (if flg (mv flg val) ,second)))

(program)
; restriction on pattern matching.
;  We don't look into quoted constants.  Quoted constants should be literally
; equal to the pattern or match to a meta-variable as it is.
; Pattern Match returns the substitution for the outer-most matching pattern.
; There may be more than two subterms that match the same pattern.
(mutual-recursion
(defun pattern-match (pattern term subst)
  (cond ((variablep pattern)
         (if (eq pattern term) (mv t subst) (mv nil nil)))
        ((fquotep pattern)
         (if (equal pattern term) (mv t subst) (mv nil nil)))
        ((fmeta-varp pattern)
         (let ((inst (assoc-eq (fmeta-var-name pattern) subst)))
           (if inst
               (if (equal term (cdr inst)) (mv t subst) (mv nil nil))
               (mv t (cons (cons (fmeta-var-name pattern) term) subst)))))
        ((and (not (variablep term))
              (not (fquotep term))
              (eq (ffn-symb pattern) (ffn-symb term)))
         (pattern-match-lst (fargs pattern) (fargs term) subst))
        (t (mv nil nil))))

(defun pattern-match-lst (patterns terms subst)
  (cond ((and (null patterns) (null terms))
         (mv t subst))
        ((or (null patterns) (null terms)) (mv nil nil))
        (t (mv-let (flg new-subst)
                   (pattern-match (car patterns) (car terms) subst)
             (if flg
                 (pattern-match-lst (cdr patterns) (cdr terms) new-subst)
                 (mv nil nil))))))
)


(mutual-recursion
(defun pattern-occur (pattern term subst)
  (if (or (variablep term) (fquotep term))
      (pattern-match pattern term subst)
      (mv2-or (pattern-match pattern term subst)
              (pattern-occur-lst pattern (fargs term) subst))))

(defun pattern-occur-lst (patterns args subst)
  (cond ((null args) (mv nil nil))
        (t (mv2-or (pattern-occur patterns (car args) subst)
                   (pattern-occur-lst patterns (cdr args) subst)))))
)

(mutual-recursion
(defun subst-meta (pattern subst)
  (cond ((or (variablep pattern) (fquotep pattern))
         pattern)
```

```
        ((fmeta-varp pattern)
         (let ((inst (assoc-eq (fmeta-var-name pattern) subst)))
           (if inst (cdr inst) pattern)))
        (t (cons (ffn-symb pattern) (subst-meta-lst (fargs pattern) subst)))))

(defun subst-meta-lst (patterns subst)
  (if (null patterns)
      nil
      (cons (subst-meta (car patterns) subst)
            (subst-meta-lst (cdr patterns) subst))))
)

(defmacro when-pattern-found (term hint)
  `(mv-let (flg subst) (pattern-occur-lst ',term clause nil)
    (if flg (subst-meta ',hint subst) nil)))

(defun multiple-pattern-check (terms)
  (if (endp terms)
      nil
      (if (endp (cdr terms))
          `(pattern-occur-lst ',(car terms) clause nil)
          `(mv-let (flg subst) ,(multiple-pattern-check (cdr terms))
            (if flg
                (pattern-occur-lst ',(car terms) clause subst)
                (mv nil nil))))))


(defmacro when-multi-pattern-found (terms hint)
  `(mv-let (flg subst) ,(multiple-pattern-check (reverse terms))
    (if flg (subst-meta ',hint subst) nil)))
```

D.1.6 b-ops-aux-def.lisp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; b-ops-aux-def.lisp
; Author  Jun Sawada, University of Texas at Austin
;
; This book contains definitions of auxiliary definition to the IHS.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(in-package "ACL2")

(include-book "trivia")
(include-book "ihs")

(deflabel begin-b-ops-aux-def)

;; Bs-and takes arbitrary number of bits. It returns 1 if all
;; arguments are 1's, otherwise return 0.
(defmacro bs-and (x y &rest rst)
  (xxxjoin 'b-and (cons x (cons y rst))))

;; Bs-and takes arbitrary number of bits. It returns 1 if any
;; argument is 1, otherwise return 0.
(defmacro bs-ior (x y &rest rst)
  (xxxjoin 'b-ior (cons x (cons y rst))))

;; b1p returns T if x is 1, nil if x is 0.
(defun b1p (x)
  (declare (xargs :guard (bitp x)))
```

```
  (not (zbp x)))

;; Bit if operation.
(defmacro b-if (test a1 ax)
  `(if (b1p ,test) ,a1 ,ax))

;; N-bit bit vector comparator.  If the least significant n-bit of
;; vectors x and y are equal, bv-eqv returns 1.
(defun bv-eqv (n x y)
  (declare (xargs :guard (and (integerp x) (integerp y)
                              (integerp n) (<= 0 n))))
  (if (equal (loghead n x) (loghead n y)) 1 0))

(defthm bitp-bv-eqv
    (bitp (bv-eqv n x y)))

(defthm bit-p-unsigned-byte-p-1
            (implies (bitp x)
                     (unsigned-byte-p 1 x))
          :hints (("Goal" :in-theory (enable unsigned-byte-p bitp))))

;; Note on disabled and enabled functions
;; All bit operations except for b-if are disabled.  We'd like to
;; deal with bit operations without opening them.  However, enabling
;; b-if allows the prover to examine cases automatically.

(in-theory (disable b1p))
(in-theory (disable bv-eqv))
```

D.1.7 b-ops-aux.lisp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; b-ops-aux.lisp:
; Author  Jun Sawada, University of Texas at Austin
;
;  This book contains auxiliary lemmas to assist the IHS library
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(in-package "ACL2")

(include-book "b-ops-aux-def")

(deflabel begin-b-ops-aux)

;; Type of b-if
(defthm bitp-b-if
    (implies (and (bitp y) (bitp z))
             (bitp (b-if x y z)))
  :hints (("goal" :in-theory (enable bitp))))

(defthm integerp-b-if
    (implies (and (integerp y) (integerp z))
             (integerp (b-if x y z)))
  :hints (("goal" :in-theory (enable bitp))))

(defthm b-not-b-not
    (equal (b-not (b-not i)) (bfix i))
  :hints (("goal" :in-theory (enable b-not bfix zbp))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Here are several basic lemmas about bit operations.
```

```
;; If bit vector X is longer than (p+s), bits between the p'th and (p+s)'th
;; bits of the concatenation of x and y is equal to that of x.
(defthm rdb-logapp-1
    (implies (and (integerp x) (integerp y) (integerp i) (<= 0 i)
                  (integerp s) (<= 0 s) (integerp p) (<= 0 p)
                  (<= (+ s p) i))
             (equal (rdb (cons s p) (logapp i x y))
                    (rdb (cons s p) x)))
  :hints (("Goal" :in-theory (enable logapp* rdb bsp-size bsp-position))))

;; If bit vector X is shorter than p, bits from the p'th bit to the
;; (p+s)'th bit of the concatenation of x and y is equal to that of y.
(defthm rdb-logapp-2
    (implies (and (integerp x) (integerp y)
                  (integerp s) (<= 0 s) (integerp p) (<= 0 p)
                  (integerp i) (<= 0 i)
                  (<= i p))
             (equal (rdb (cons s p) (logapp i x y))
                    (rdb (cons s (- p i)) y)))
  :hints (("Goal" :in-theory (enable logapp* rdb bsp-size bsp-position))))

(defthm loghead-0
    (equal (loghead x 0) 0)
  :hints (("Goal" :in-theory (enable loghead))))

(defthm loghead-1
    (equal (loghead 1 vector) (logcar vector))
  :hints (("Goal" :in-theory (enable logcar loghead))))

(defthm logcar-bitp
    (implies (bitp x)
             (equal (logcar x) x))
  :hints (("Goal" :in-theory (enable bitp logcar))))

(defthm logbit-0-bitp
    (implies (bitp x)
             (equal (logbit 0 x) x))
    :hints (("Goal" :in-theory (enable bitp logbit))))

(defthm loghead-bitp
    (implies (bitp x)
             (equal (loghead 1 x) x))
  :hints (("Goal" :in-theory (enable rdb))))

(defthm logcons-0
    (implies (bitp x)
             (equal (logcons x 0) x))
  :hints (("Goal" :in-theory (enable logcons))))

(defthm rdb-bitp
    (implies (bitp x)
             (equal (rdb (cons 1 0) x) x))
  :hints (("Goal" :in-theory (enable rdb))))

(defthm rdb-0
    (equal (rdb (cons 0 n) x) 0)
  :hints (("Goal" :in-theory (enable rdb bsp-size))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Here begins a basic theory about lobops
; Recursive definition of Logbit*.
```

```
(defthm logbit*
    (implies (and (integerp pos)
                  (>= pos 0)
                  (integerp i))
             (equal (logbit pos i)
                    (if (equal pos 0)
                        (logcar i)
                        (logbit (1- pos) (logcdr i)))))
  :rule-classes :definition
  :hints (("Goal" :in-theory (enable logbit logbitp* logbitp))))

(in-theory (disable logbit*))

(local
(defun logtail-induct (pos i)
  (if (zp pos)
      i
      (logtail-induct (1- pos) (logcdr i)))))

(defthm logcar-logtail
    (implies (and (integerp n) (<= 0 n)
                  (integerp x))
             (equal (logcar (logtail n x))
                    (logbit n x)))
  :hints (("Goal" :in-theory (enable logtail* logbit logbitp*)
                  :induct (logtail-induct n x))))

(defthm logcdr-logtail
    (implies (and (integerp n) (<= 0 n)
                  (integerp x))
             (equal (logcdr (logtail n x))
                    (logtail (1+ n) x)))
  :hints (("Goal" :in-theory (enable logtail*)
                   :induct (logtail-induct n x))))

;; The following two lemmas are for expansion before BDD.
(deflabel begin-bv-expand)
(defthm rdb-expand
   (implies (and (syntaxp (and (quoted-constant-p s) (quoted-constant-p n)))
                 (integerp s) (< 0 s)
                 (integerp n) (<= 0 n)
                 (integerp x))
            (equal (rdb (cons s n) x)
                   (logcons (logbit n x) (rdb (cons (1- s) (1+ n)) x))))
  :hints (("Goal" :in-theory (enable rdb bsp-position bsp-size loghead*
                                     logtail*))))

;; logops supplement for logxxx operations. There is a bunch of
;; definition rules such as logior*.  We apply rules to forcibly open
;; up the expressions before applying BDD techniques.
(defthm open-logior-right-const
    (implies (and (bitp i1)
                  (integerp i2)
                  (integerp j))
             (equal (logior (logcons i1 i2) j)
                    (logcons (b-ior i1 (logcar j)) (logior i2 (logcdr j)))))
  :hints (("Goal" :in-theory (enable logior*))))

(defthm open-logior-left-const
    (implies (and (bitp j1)
                  (integerp j2)
                  (integerp i))
```

```
            (equal (logior i (logcons j1 j2))
                   (logcons (b-ior (logcar i) j1) (logior (logcdr i) j2))))
  :hints (("Goal" :in-theory (enable logior*))))

(defthm open-logior
    (implies (and (bitp i1)
                  (bitp j1)
                  (integerp i2)
                  (integerp j2))
             (equal (logior (logcons i1 i2) (logcons j1 j2))
                    (logcons (b-ior i1 j1) (logior i2 j2))))
  :hints (("Goal" :in-theory (enable logior*))))

(defthm open-logxor-right-const
    (implies (and (bitp i1)
                  (integerp i2)
                  (integerp j))
             (equal (logxor (logcons i1 i2) j)
                    (logcons (b-xor i1 (logcar j)) (logxor i2 (logcdr j)))))
  :hints (("Goal" :in-theory (enable logxor*))))

(defthm open-logxor-left-const
    (implies (and (bitp j1)
                  (integerp j2)
                  (integerp i))
             (equal (logxor i (logcons j1 j2))
                    (logcons (b-xor (logcar i) j1) (logxor (logcdr i) j2))))
  :hints (("Goal" :in-theory (enable logxor*))))

(defthm open-logxor
    (implies (and (bitp i1)
                  (bitp j1)
                  (integerp i2)
                  (integerp j2))
             (equal (logxor (logcons i1 i2) (logcons j1 j2))
                    (logcons (b-xor i1 j1) (logxor i2 j2))))
  :hints (("Goal" :in-theory (enable logxor*))))

(defthm open-logand-right-const
    (implies (and (bitp i1)
                  (integerp i2)
                  (integerp j))
             (equal (logand (logcons i1 i2) j)
                    (logcons (b-and i1 (logcar j)) (logand i2 (logcdr j)))))
  :hints (("Goal" :in-theory (enable logand*))))

(defthm open-logand-left-const
    (implies (and (bitp j1)
                  (integerp j2)
                  (integerp i))
             (equal (logand i (logcons j1 j2))
                    (logcons (b-and (logcar i) j1) (logand (logcdr i) j2))))
  :hints (("Goal" :in-theory (enable logand*))))

(defthm open-logand
    (implies (and (bitp i1)
                  (bitp j1)
                  (integerp i2)
                  (integerp j2))
             (equal (logand (logcons i1 i2) (logcons j1 j2))
                    (logcons (b-and i1 j1) (logand i2 j2))))
  :hints (("Goal" :in-theory (enable logand*))))
```

```
(defthm open-lognot
    (implies (and (bitp i1)
                  (integerp i2))
             (equal (lognot (logcons i1 i2))
                    (logcons (b-not i1) (lognot i2))))
  :hints (("Goal" :in-theory (enable lognot*))))

(defthm fold-logcdr-vector
    (implies (and (integerp x)
                  (syntaxp (or (not (consp x))
                               (not (equal (car x) 'logcons)))))
             (equal (logcdr x) (logtail 1 x)))
  :hints (("goal" :in-theory (enable logtail*))))
(in-theory (disable fold-logcdr-vector))

(defthm fold-logcar-vector
    (implies (and (integerp x)
                  (syntaxp (or (not (consp x))
                               (not (equal (car x) 'logcons)))))
             (equal (logcar x) (logbit 0 x)))
  :hints (("goal" :in-theory (enable logbit logbitp*))))
(in-theory (disable fold-logcar-vector))
(deflabel end-bv-expand)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Bv-eqv rules
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;x
(defthm bv-eqv*
    (implies (and (integerp x)
                  (integerp y)
                  (integerp n)
                  (>= n 0))
             (equal (bv-eqv n x y)
                    (if (zp n)
                        1
                        (b-and (b-eqv (logcar x) (logcar y))
                               (bv-eqv (1- n) (logcdr x) (logcdr y))))))
  :hints (("goal" :in-theory (enable bv-eqv loghead*)))
  :rule-classes :definition)

(in-theory (disable bv-eqv*))

;; Following lemma is useful in converting a formula with 'bv-eqv' to
;; one with 'equal'.  This can cause BDD simplification to fail,
;; especially if when the words need to be open up.
(defthm bv-eqv-iff-equal
  (implies (and (unsigned-byte-p n x) (unsigned-byte-p n y))
           (equal (b1p (bv-eqv n x y))
                  (equal x y)))
  :hints (("Goal" :in-theory (enable bv-eqv b1p))))

(defthm bv-eqv-0
    (equal (bv-eqv 0 x y) 1)
  :Hints (("Goal" :in-theory (enable bv-eqv))))

(defthm bv-x-x
    (equal (bv-eqv n x x) 1)
  :hints (("goal" :in-theory (enable bv-eqv))))

(defthm bv-eqv-assoc
    (equal (bv-eqv n x y) (bv-eqv n y x))
```

```
  :hints (("Goal" :in-theory (enable bv-eqv))))

(defthm bv-eqv-bits
    (implies (and (bitp x) (bitp y) (integerp n) (> n 0))
             (equal (bv-eqv n x y) (b-eqv x y)))
  :hints (("Goal" :in-theory (enable bv-eqv* bitp bv-eqv-iff-equal))))

(defthm bv-eqv-expand
    (implies (and (integerp x) (integerp y)
                  (integerp n) (> n 0)
                  (syntaxp (quoted-constant-p n)))
             (equal (bv-eqv n x y)
                    (b-and (b-eqv (logcar x) (logcar y))
                           (bv-eqv (1- n) (logcdr x) (logcdr y)))))
  :hints (("Goal" :in-theory (enable bv-eqv*))))

;; Following lemma is a schematic lemma which will be used in BDD proofs.
;; We noticed that several lemmas is a tautology under the property that
;; (bv-eqv x y) and (bv-eqv x z) are not simultaneously asserted, if y and
;; z are not equal. We instantiate the following lemma and add it to BDD
;; proof as a hint.
(defthm bv-eqv-transitivity
    (implies (and (unsigned-byte-p n x)
                  (unsigned-byte-p n y)
                  (unsigned-byte-p n z)
                  (b1p (b-and (bv-eqv n x z) (bv-eqv n y z))))
             (equal x y))
  :hints (("Goal" :in-theory (enable bv-eqv )))
  :rule-classes nil)

;; Bv-expander is the theory to be enabled to expand bit vectors before
;; applying bdd's.
(deftheory bv-expander
    '(rdb-expand
      open-lognot
      open-logand open-logand-right-const open-logand-left-const
      open-logior open-logior-right-const open-logior-left-const
      open-logxor open-logxor-right-const open-logxor-left-const
      fold-logcar-vector fold-logcdr-vector
      bv-eqv-bits
      bv-eqv-0
      bv-eqv-expand))

(in-theory (disable bv-expander))

(local (in-theory (enable b1p)))

;; Bit-boolean-converter converts expressions of form (equal x 1) to
;; expressions using bitp.  It also converts (equal x 0) to (not (b1p
;; x)).
(defthm equal-to-1-to-b1p
    (implies (bitp x)
             (equal (equal x 1) (b1p x)))
  :Hints (("Goal" :In-theory (enable bitp))))

(defthm equal-to-0-to-not-b1p
    (implies (bitp x)
             (equal (equal x 0) (not (b1p x))))
  :Hints (("Goal" :In-theory (enable bitp))))

(defthm equal-to-b1p-b-eqv
    (implies (and (bitp x) (bitp y))
```

```
            (equal (equal x y) (b1p (b-eqv x y))))
  :hints (("Goal" :in-theory (enable b-eqv b1p zbp bitp))))

(deftheory equal-b1p-converter
    '(equal-to-1-to-b1p equal-to-0-to-not-b1p))

(in-theory (disable equal-b1p-converter))
(in-theory (disable  equal-to-b1p-b-eqv))

;; From here, we define bit-to-boolean converter, especially for BDD
;; operation.
(deflabel begin-lift-b-ops)

(defthm zbp-b-and
    (equal (zbp (b-and x y))
           (or (zbp x) (zbp y)))
  :Hints (("Goal" :in-theory (enable b-and))))

(defthm zbp-b-ior
    (equal (zbp (b-ior x y))
           (and (zbp x) (zbp y)))
  :Hints (("Goal" :in-theory (enable b-ior))))

(defthm zbp-b-xor
    (equal (zbp (b-xor x y))
           (or (and (zbp x) (zbp y))
               (and (not (zbp x)) (not (zbp y)))))
  :Hints (("Goal" :in-theory (enable b-xor))))

(defthm zbp-b-not
    (equal (zbp (b-not x))
           (not (zbp x)))
  :Hints (("Goal" :in-theory (enable b-not))))

(defthm zbp-b-eqv
    (equal (zbp (b-eqv x y))
           (not (iff (zbp x) (zbp y))))
  :Hints (("Goal" :in-theory (enable b-eqv))))

(defthm b1p-b-and
    (equal (b1p (b-and x y))
           (and (b1p x) (b1p y)))
  :Hints (("Goal" :in-theory (enable b-and))))

(defthm b1p-b-ior
    (equal (b1p (b-ior x y))
           (or (b1p x) (b1p y)))
  :Hints (("Goal" :in-theory (enable b-ior))))

(defthm b1p-b-xor
    (equal (b1p (b-xor x y))
           (or (and (b1p x) (not (b1p y)))
               (and (not (b1p x)) (b1p y))))
  :Hints (("Goal" :in-theory (enable b-xor))))

(defthm b1p-b-not
    (equal (b1p (b-not x))
           (not (b1p x)))
  :Hints (("Goal" :in-theory (enable b-not))))

(defthm b1p-b-eqv
    (equal (b1p (b-eqv x y))
```

```
          (iff (b1p x) (b1p y)))
  :Hints (("Goal" :in-theory (enable b-eqv))))

(defthm b1p-nand (equal (b1p (b-nand x y)) (not (and (b1p x) (b1p y))))
  :hints (("Goal" :in-theory (enable b-nand))))

(defthm b1p-nor (equal (b1p (b-nor x y)) (not (or (b1p x) (b1p y))))
    :hints (("Goal" :in-theory (enable b-nor))))

(defthm b1p-andc1 (equal (b1p (b-andc1 x y)) (and (not (b1p x)) (b1p y)))
  :hints (("Goal" :in-theory (enable b-andc1))))

(defthm b1p-andc2 (equal (b1p (b-andc2 x y)) (and (b1p x) (not (b1p y))))
  :hints (("Goal" :in-theory (enable b-andc2))))

(defthm b1p-orc1 (equal (b1p (b-orc1 x y)) (or (not (b1p x)) (b1p y)))
  :hints (("Goal" :in-theory (enable b-orc1))))

(defthm b1p-orc2 (equal (b1p (b-orc2 x y)) (or (b1p x) (not (b1p y))))
  :hints (("Goal" :in-theory (enable b-orc2))))

(defthm zbp-to-b1p
    (equal (zbp x) (not (b1p x)))
  :hints (("Goal" :in-theory (enable b1p))))

(deflabel end-lift-b-ops)

(deftheory lift-b-ops
    (set-difference-theories (universal-theory 'end-lift-b-ops)
                             (universal-theory 'begin-lift-b-ops)))

(in-theory (disable lift-b-ops))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; New Simplifier
; This simplifier simplifies bit terms into 1 and 0's using
; lemmas about b1p.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deflabel begin-b1p-bit-rewriter)

(defthm b1p-bit-forward
    (implies (and (b1p b) (force (bitp b))) (equal b 1))
  :hints (("goal" :in-theory (enable b1p bitp zbp)))
  :rule-classes :forward-chaining)

(defthm not-b1p-bit-forward
    (implies (and (not (b1p b)) (force (bitp b))) (equal b 0))
  :hints (("goal" :in-theory (enable b1p bitp zbp)))
  :rule-classes :forward-chaining)

(defthm simplify-bit-functions-2
    (and (implies (and (bitp x) (bitp y) (not (b1p x)))
                  (equal (b-and x y) 0))
         (implies (and (bitp x) (bitp y) (not (b1p y)))
                  (equal (b-and x y) 0))
         (implies (and (bitp x) (bitp y) (b1p x))
                  (equal (b-and x y) y))
         (implies (and (bitp x) (bitp y) (b1p y))
                  (equal (b-and x y) x))

         (implies (and (bitp x) (bitp y) (not (b1p x)))
                  (equal (b-ior x y) y))
```

```
(implies (and (bitp x) (bitp y) (not (b1p y)))
         (equal (b-ior x y) x))
(implies (and (bitp x) (bitp y) (b1p x))
         (equal (b-ior x y) 1))
(implies (and (bitp x) (bitp y) (b1p y))
         (equal (b-ior x y) 1))

(implies (and (bitp x) (bitp y) (not (b1p x)))
         (equal (b-xor x y) y))
(implies (and (bitp x) (bitp y) (not (b1p y)))
         (equal (b-xor x y) x))
(implies (and (bitp x) (bitp y) (b1p x))
         (equal (b-xor x y) (b-not y)))
(implies (and (bitp x) (bitp y) (b1p y))
         (equal (b-xor x y) (b-not x)))

(implies (and (bitp x) (bitp y) (not (b1p x)))
         (equal (b-eqv x y) (b-not y)))
(implies (and (bitp x) (bitp y) (not (b1p y)))
         (equal (b-eqv x y) (b-not x)))
(implies (and (bitp x) (bitp y) (b1p x))
         (equal (b-eqv x y) y))
(implies (and (bitp x) (bitp y) (b1p y))
         (equal (b-eqv x y) x))

(implies (and (bitp x) (bitp y) (not (b1p x)))
         (equal (b-nand x y) 1))
(implies (and (bitp x) (bitp y) (not (b1p y)))
         (equal (b-nand x y) 1))
(implies (and (bitp x) (bitp y) (b1p x))
         (equal (b-nand x y) (b-not y)))
(implies (and (bitp x) (bitp y) (b1p y))
         (equal (b-nand x y) (b-not x)))

(implies (and (bitp x) (bitp y) (not (b1p x)))
         (equal (b-nor x y) (b-not y)))
(implies (and (bitp x) (bitp y) (not (b1p y)))
         (equal (b-nor x y) (b-not x)))
(implies (and (bitp x) (bitp y) (b1p x))
         (equal (b-nor x y) 0))
(implies (and (bitp x) (bitp y) (b1p y))
         (equal (b-nor x y) 0))

(implies (and (bitp x) (bitp y) (not (b1p x)))
         (equal (b-andc1 x y) y))
(implies (and (bitp x) (bitp y) (not (b1p y)))
         (equal (b-andc1 x y) 0))
(implies (and (bitp x) (bitp y) (b1p x))
         (equal (b-andc1 x y) 0))
(implies (and (bitp x) (bitp y) (b1p y))
         (equal (b-andc1 x y) (b-not x)))

(implies (and (bitp x) (bitp y) (not (b1p x)))
         (equal (b-andc2 x y) 0))
(implies (and (bitp x) (bitp y) (not (b1p y)))
         (equal (b-andc2 x y) x))
(implies (and (bitp x) (bitp y) (b1p x))
         (equal (b-andc2 x y) (b-not y)))
(implies (and (bitp x) (bitp y) (b1p y))
         (equal (b-andc2 x y) 0))

(implies (and (bitp x) (bitp y) (not (b1p x)))
```

```
                (equal (b-orc1 x y) 1))
        (implies (and (bitp x) (bitp y) (not (b1p y)))
                (equal (b-orc1 x y) (b-not x)))
        (implies (and (bitp x) (bitp y) (b1p x))
                (equal (b-orc1 x y) y))
        (implies (and (bitp x) (bitp y) (b1p y))
                (equal (b-orc1 x y) 1))

        (implies (and (bitp x) (bitp y) (not (b1p x)))
                (equal (b-orc2 x y) (b-not y)))
        (implies (and (bitp x) (bitp y) (not (b1p y)))
                (equal (b-orc2 x y) 1))
        (implies (and (bitp x) (bitp y) (b1p x))
                (equal (b-orc2 x y) 1))
        (implies (and (bitp x) (bitp y) (b1p y))
                (equal (b-orc2 x y) x)))
  :hints (("Goal" :in-theory (enable b1p zbp b-and bitp))))

(deflabel end-b1p-bit-rewriter)

(deftheory b1p-bit-rewriter
    (set-difference-theories (universal-theory 'end-b1p-bit-rewriter)
                             (universal-theory 'begin-b1p-bit-rewriter)))

(in-theory (disable b1p-bit-rewriter))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Theories used for the proof using BDD.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deftheory pre-bdd-disables
    '(bv-eqv-iff-equal simplify-bit-functions))

(deftheory bdd-disables
    '(bv-eqv-iff-equal simplify-bit-functions))

(theory-invariant (incompatible (:rewrite bv-eqv-iff-equal)
                                (:rewrite rdb-expand)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; range of unsigned-byte-p
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; theories about the range of unsigned-byte-p
(defthm plus-unsigned-byte-p-lt-2-*-expt-2-width
    (implies (and (unsigned-byte-p width val1)
                  (unsigned-byte-p width val2))
             (< (+ val1 val2) (* 2 (expt 2 width))))
  :hints (("goal" :in-theory (enable unsigned-byte-p))))

(defun logbit-induction (pos val)
    (if (zp pos)
        val
        (logbit-induction (1- pos) (logcdr val))))

; If a value are in the range  0 <= val < 2^width
; then the width'th bit of val is not set.
(defthm logbit-0-if-val-lt-expt-2-width
    (implies (and (integerp width) (<= 0 width)
                  (integerp val)
                  (<= 0 val) (< val (expt 2 width)))
             (equal (logbit width val) 0))
  :hints (("goal" :in-theory (e/d (logbit* expt logcar logcdr)
                                  (exponents-add))
```

```
                 :induct (logbit-induction width val)))
  :rule-classes nil)

(encapsulate nil
(local
 (defthm logbit-1-if-val-gt-expt-2-width-help1
     (IMPLIES (AND (INTEGERP WIDTH)
                   (< 0 WIDTH)
                   (INTEGERP VAL)
                   (<= (* 2 (EXPT 2 (+ -1 WIDTH))) VAL)
                   (< (/ VAL 2) (* 2 (EXPT 2 (+ -1 WIDTH)))))
              (> (* 2 (EXPT 2 (+ -1 WIDTH)))
                 (FLOOR VAL 2)))
 :rule-classes nil))

(local
 (defthm logbit-1-if-val-gt-expt-2-width-help2
     (implies  (and (integerp val)
                    (integerp width) (< 0 width)
                    (< VAL (* 2 2 (EXPT 2 (+ -1 WIDTH)))))
               (< (/ VAL 2) (* 2 (EXPT 2 (+ -1 WIDTH)))))
   :hints (("goal" :in-theory (e/d (expt) (exponents-add))))
   :rule-classes nil))

(local
(defthm logbit-1-if-val-gt-expt-2-width-help
    (IMPLIES (AND (INTEGERP WIDTH)
                  (INTEGERP VAL)
                  (< 0 WIDTH)
                  (<= (* 2 (EXPT 2 (+ -1 WIDTH))) (FLOOR VAL 2)))
             (<= (* 2 2 (EXPT 2 (+ -1 WIDTH))) VAL))
  :hints (("goal" :in-theory (e/d (exponents-add) (expt))
                  :use ((:instance logbit-1-if-val-gt-expt-2-width-help1)
                        (:instance logbit-1-if-val-gt-expt-2-width-help2))))))

; If a value are in the range  2^width <= val < 2^width * 2,
; then the width'th bit of val is set.
(defthm logbit-1-if-val-gt-expt-2-width
    (implies (and (integerp width) (<= 0 width)
                  (integerp val)
                  (<= (expt 2 width) val)
                  (< val (* 2 (expt 2 width))))
             (equal (logbit width val) 1))
  :hints (("goal" :in-theory (e/d (logbit* expt logcar logcdr)
                                  (exponents-add))
                  :induct (logbit-induction width val)))
  :rule-classes nil)
)

; Suppose val1 and val2 are unsigned-byte-p whose width is w.
; If w'th bit of the sum (+ val1 val2) is not set,
; (+ val1 val2) < 2^w.
(defthm plus-unsigned-byte-lt-expt-2-width-if-logbit
    (implies (and (unsigned-byte-p width val1)
                  (unsigned-byte-p width val2)
                  (not (b1p (logbit width (+ val1 val2)))))
             (< (+ val1 val2) (expt 2 width)))
  :hints (("goal" :in-theory (enable unsigned-byte-p)
                  :use (:instance logbit-1-if-val-gt-expt-2-width
                                  (val (+ val1 val2))))))

; Suppose val1 and val2 are unsigned-byte-p whose width is w.
```

```
; If w'th bit of the sum (+ val1 val2) is set, then
;   2^w < (+ val1 val2).
(defthm plus-unsigned-byte-gt-expt-2-width-if-logbit
    (implies (and (unsigned-byte-p width val1)
                  (unsigned-byte-p width val2)
                  (b1p (logbit width (+ val1 val2))))
             (<= (expt 2 width) (+ val1 val2)))
  :hints (("goal" :in-theory (enable unsigned-byte-p)
                  :use (:instance logbit-0-if-val-lt-expt-2-width
                                  (val (+ val1 val2))))))

; Suppose val1 and val2 are unsigned-byte-p whose width is w.
; If the sum of val1 and val2 does not carry out to w'th bit,
;  (loghead w (+ val1 val2)) = (+ val1 val2)
(defthm loghead-unsigned-byte-+-if-not-carry
    (implies (and (integerp width)
                  (<= 0 width)
                  (unsigned-byte-p width val1)
                  (unsigned-byte-p width val2)
                  (not (b1p (logbit width (+ val1 val2)))))
             (equal (loghead width (+ val1 val2)) (+ val1 val2)))
  :hints (("goal" :in-theory (enable loghead)
                  :do-not-induct t)))

(encapsulate nil
(local
(defthm j*k-ge-2*k-if-j-gt-1
    (implies (and (integerp j)
                  (< 1 j)
                  (integerp k)
                  (< 0 k))
             (<= (* 2 k) (* j k)))
  :hints (("Goal" :in-theory (enable <-*-right-cancel)))
  :rule-classes :linear))

; A trivia theorem.
; If y < x < 2*y, then (x mod y) = x - y
(defthm mod-x-y-=-minus-x-y
 (implies (and (integerp x) (integerp y) (< 0 y)
               (<= y x) (< x (* 2 y)))
          (equal (mod x y) (- x y)))
 :Hints (("goal" :in-theory (disable (:generalize floor-bounds)))
         ("subgoal 1" :cases ((<= j 1)))))
)

(in-theory (disable mod-x-y-=-minus-x-y))

; Suppose val1 and val2 are unsigned-byte-p whose width is w.
; If the sum of val1 and val2 does not carry out to w'th bit,
;  (loghead w (+ val1 val2)) = (+ val1 val2)
(defthm loghead-unsigned-byte-+-if-carry
    (implies (and (integerp width)
                  (<= 0 width)
                  (unsigned-byte-p width val1)
                  (unsigned-byte-p width val2)
                  (b1p (logbit width (+ val1 val2))))
             (equal (loghead width (+ val1 val2))
                    (- (+ val1 val2) (expt 2 width))))
  :hints (("goal" :in-theory (enable loghead mod-x-y-=-minus-x-y)
                  :do-not-induct t)))
```

D.2 Machine Definitions

This chapter contains the definition of the FM9801. See Chapter 5 for discussions.

D.2.1 basic-def.lisp

This section provide the definition of data words, address words, register files containing general-purpose registers, special registers, and the memory. These definitions are used in the ISA and MA definitions of the FM9801.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Basic-def.lisp:
; Author  Jun Sawada, University of Texas at Austin
;
; This file includes various underlying definition for the ISA and
; MA designs.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(in-package "ACL2")

(include-book "absolute-path")
(include-acl2-book "data-structures/array1")
(include-acl2-book "data-structures/deflist")
(include-acl2-book "data-structures/list-defthms")
(include-acl2-book "data-structures/structures")
(include-book "ihs")

(include-book "trivia")
(include-book "b-ops-aux")

(deflabel begin-basic-def)

(defmacro naturalp (x) `(and (integerp ,x) (<= 0 ,x)))
(defconst *addr-size* 16)
(defconst *page-offset-size* 10)

(defconst *rname-size* 4)
(defconst *immediate-size* 8)
(defconst *opcode-size* 4)
(defconst *word-size* 16)
(defconst *max-word-value* (expt 2 *word-size*))
(defconst *num-regs* (expt 2 *rname-size*))
(defconst *num-mem-words* (expt 2 *addr-size*))
(defconst *num-page-words* (expt 2 *page-offset-size*))
(defconst *num-pages* (expt 2 (- *addr-size* *page-offset-size*)))
(defbytetype word *word-size* :unsigned)
(defbytetype addr  *addr-size* :unsigned)
(defbytetype rname *rname-size* :unsigned)
(defbytetype immediate *immediate-size* :unsigned)
(defbytetype opcd  *opcode-size* :unsigned)

(defthm word-p-type
    (implies (word-p word)
             (and (integerp word)
                  (>= word 0)
```

```
                (< word *max-word-value*)))
  :hints (("Goal" :in-theory (enable word-p)))
  :rule-classes :forward-chaining)

(defthm word-p-type-def
    (iff (word-p x)
         (and (integerp x)
              (<= 0 x) (< x (expt 2 16))))
  :hints (("goal" :in-theory (enable word-p unsigned-byte-p)))
  :rule-classes nil)

(defthm word-mod
    (implies (integerp x)
             (equal (word x)
                    (mod x (expt 2 16))))
  :hints (("goal" :in-theory (enable word loghead)))
  :rule-classes nil)

(defthm word-idem
    (implies (word-p x) (equal (word x) x))
  :rule-classes nil)

(defthm rname-p-type
    (implies (rname-p rname)
             (and (integerp rname)
                  (>= rname 0)
                  (< rname *num-regs*)))
  :hints (("Goal" :in-theory (enable rname-p)))
  :rule-classes :forward-chaining)

(defthm addr-p-type-def
    (iff (addr-p x)
         (and (integerp x)
              (<= 0 x) (< x (expt 2 16))))
  :hints (("goal" :in-theory (enable addr-p unsigned-byte-p)))
  :rule-classes nil)

(defthm addr-mod
    (implies (integerp x)
             (equal (addr x)
                    (mod x (expt 2 16))))
  :hints (("goal" :in-theory (enable addr loghead)))
  :rule-classes nil)

(defthm addr-idem
    (implies (addr-p x) (equal (addr x) x))
  :rule-classes nil)

(defthm addr-p-type
    (implies (addr-p ad)
             (and (integerp ad)
                  (>= ad  0)
                  (< ad *num-mem-words*)))
  :hints (("Goal" :in-theory (enable addr-p)))
  :rule-classes :forward-chaining)

(defthm addr-word-double-casting
    (equal (addr (word i)) (addr i))
  :hints (("goal" :in-theory (enable addr word))))

(defthm word-addr-double-casting
    (equal (word (addr i)) (word i))
```

```
  :hints (("goal" :in-theory (enable addr word))))

(defthm word-addr-p
    (implies (addr-p x) (equal (word x) x))
  :hints (("goal" :in-theory (enable addr-p word))))

(defthm addr-word-p
    (implies (word-p x) (equal (addr x) x))
  :hints (("goal" :in-theory (enable addr word-p))))

(in-theory (disable word-addr-p addr-word-p))

(defthm word-p-logand
    (implies (and (word-p val1) (word-p val2))
             (word-p (logand val1 val2)))
  :hints (("Goal" :in-theory (enable word-p))))

(defthm word-p-logxor
    (implies (and (word-p val1) (word-p val2))
             (word-p (logxor val1 val2)))
  :hints (("Goal" :in-theory (enable word-p))))

(defthm word-p-logior
    (implies (and (word-p val1) (word-p val2))
             (word-p (logior val1 val2)))
  :hints (("Goal" :in-theory (enable word-p))))

(defthm word-p-bv-eqv-iff-equal
    (implies (and (word-p wd0) (word-p wd1))
             (equal (b1p (bv-eqv *word-size* wd0 wd1)) (equal wd0 wd1)))
  :hints (("Goal" :in-theory (enable word-p))))

(defthm addr-p-bv-eqv-iff-equal
    (implies (and (addr-p wd0) (addr-p wd1))
             (equal (b1p (bv-eqv *addr-size* wd0 wd1)) (equal wd0 wd1)))
  :hints (("Goal" :in-theory (enable addr-p))))

(defthm rname-p-bv-eqv-iff-equal
    (implies (and (rname-p wd0) (rname-p wd1))
             (equal (b1p (bv-eqv *rname-size* wd0 wd1)) (equal wd0 wd1)))
  :hints (("Goal" :in-theory (enable rname-p))))

(defthm immediate-p-bv-eqv-iff-equal
    (implies (and (immediate-p wd0) (immediate-p wd1))
             (equal (b1p (bv-eqv *immediate-size* wd0 wd1)) (equal wd0 wd1)))
  :hints (("Goal" :in-theory (enable immediate-p))))

(defthm opcd-p-bv-eqv-iff-equal
    (implies (and (opcd-p wd0) (opcd-p wd1))
             (equal (b1p (bv-eqv *opcode-size* wd0 wd1)) (equal wd0 wd1)))
  :hints (("Goal" :in-theory (enable opcd-p))))

(deflist word-listp (l)
  (declare (xargs :guard t))
  word-p)

(defun fixlen-word-listp (n lst)
  "test if list is a array of n words."
  (declare (xargs :guard (and (integerp n) (<= 0 n))))
  (and (word-listp lst) (equal (len lst) n)))

(defthm fixlen-word-true-listp
```

```
    (implies (fixlen-word-listp n x)
             (true-listp x))
  :rule-classes :forward-chaining)

(in-theory (disable fixlen-word-listp))

; Register file is a fixed length true list of words.
(defun RF-p (RF)
  (declare (xargs :guard t))
  (fixlen-word-listp *num-regs* RF))

(defthm RF-p-true-listp
    (implies (RF-p x)
             (true-listp x))
  :rule-classes :forward-chaining)

(defun read-reg (r RF)
  (declare (xargs :guard (and (rname-p r) (RF-p RF))))
  (nth r RF))

(defun write-reg (val r RF)
  (declare (xargs :guard (and (word-p val) (rname-p r) (RF-p RF))))
  (update-nth r val RF))

(defthm RF-p-write-reg
    (implies (and (word-p word)
                  (rname-p rname)
                  (RF-p RF))
             (RF-p (write-reg word rname RF)))
  :hints (("Goal" :in-theory (enable RF-p fixlen-word-listp
                                     rname-p UNSIGNED-BYTE-P
                                     len-update-nth))))

(local
(defthm nth-content-word-listp
    (implies (and (integerp n)
                  (<= 0 n)
                  (word-listp lst)
                  (< n (len lst)))
             (and (integerp (nth n lst))
                  (acl2-numberp (nth n lst))))
  :hints (("Goal" :use ((:instance word-listp-nth (n0 n) (l lst)))
                  :in-theory (disable word-listp-nth)))))

(defthm numberp-read-reg
    (implies (and (RF-p RF)
                  (rname-p rname))
             (and (integerp (read-reg rname RF))
                  (acl2-numberp (read-reg rname RF))))
  :hints (("Goal" :in-theory (enable rname-p RF-p fixlen-word-listp)))
  :rule-classes
  ((:type-prescription) (:rewrite)))

(defthm word-p-read-reg
    (implies (and (RF-p RF)
                  (rname-p rname))
             (word-p (read-reg rname RF)))
  :hints (("Goal" :in-theory (enable rname-p RF-p fixlen-word-listp))))

(defthm read-reg-write-reg
    (implies (and (rname-p r1)
                  (rname-p r2)
```

```
                (RF-p RF))
           (equal (read-reg r1 (write-reg val r2 RF))
                  (if (equal r1 r2) val (read-reg r1 RF))))
  :hints (("Goal" :in-theory (enable read-reg write-reg RF-p
                                     nth-update-nth))))

(in-theory (disable RF-p write-reg read-reg))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Here we define the memory system.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defconst *no-access* 0)
(defconst *read-only* 1)
(defconst *read-write* 2)

;;  Definition of Address Transformers.
;;  page-num Address --> Page number
;;  page-offset Address --> Page Offset
(defun page-num (addr)
  (declare (xargs :guard (addr-p addr)))
  (floor addr *num-page-words*))

(defun page-offset (addr)
  (declare (xargs :guard (addr-p addr)))
  (mod addr *num-page-words*))

(defthm page-num-type
    (implies (addr-p addr)
             (and (integerp (page-num addr))
                  (<= 0 (page-num addr))))
  :rule-classes :type-prescription)

(defthm page-num-bound
    (implies (addr-p addr)
             (< (page-num addr) *num-pages*))
  :hints (("Goal" :in-theory (enable addr-p unsigned-byte-p)))
  :rule-classes :linear)

(defthm page-offset-type
    (implies (addr-p addr)
             (and (integerp (page-offset addr))
                  (<= 0 (page-offset addr))))
  :rule-classes :type-prescription)

(defthm page-offset-bound
    (implies (addr-p addr)
             (< (page-offset addr) *num-page-words*))
  :rule-classes :linear)

(in-theory (disable page-num page-offset))
;;  End of Address Transformers

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Definition of Memory Object
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Check if page is an array representing a page.
; Addr does not affect the value, but if the correct page tag is given,
; execution is faster.
(defun word-array-p (array)
  (declare (xargs :guard (alistp array)
                  :verify-guards nil))
  (cond ((endp array) t)
```

```
        ((equal (caar array) ':header) (word-array-p (cdr array)))
        (t (and (word-p (cdar array))
                (word-array-p (cdr array))))))

(verify-guards word-array-p
               :hints (("Goal" :in-theory (enable alistp))))

(defun page-array-p (tag page-array)
  (declare (xargs :guard t))
  (and (array1p tag page-array)
       (equal (car (dimensions tag page-array)) *num-page-words*)
       (word-p (default tag page-array))
       (word-array-p page-array)))

(defstructure page
  (tag (:assert (symbolp tag) :rewrite))
  (mode (:assert (integerp mode) :rewrite))
  (array (:assert (page-array-p tag array) :rewrite))
  (:options :guards))

(defun mem-array-p (array)
  (declare (xargs :guard (alistp array)
                  :verify-guards nil))
  (cond ((endp array) t)
        ((equal (caar array) ':header) (mem-array-p (cdr array)))
        (t (and (let ((page (cdar array)))
                  (if (integerp page)
                      (or (equal page *no-access*) (equal page *read-only*)
                          (equal page *read-write*))
                      (page-p page)))
                (mem-array-p (cdr array))))))

(verify-guards mem-array-p :hints (("Goal" :in-theory (enable alistp))))

(defun mem-p (mem)
  (declare (xargs :guard t))
  (and (array1p 'mem mem)
       (equal (car (dimensions 'mem mem)) *num-pages*)
       (equal (default 'mem mem) *no-access*)
       (mem-array-p mem)))

(in-theory (disable word-array-p page-array-p mem-array-p mem-p))

(defthm page-p-type
    (implies (page-p p)
             (consp p))
  :hints (("Goal" :in-theory (enable page-p)))
  :rule-classes :compound-recognizer)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Definition of Read-mem
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defmacro get-page (n mem)
  `(aref1 'mem ,mem ,n))

(defmacro set-page (page n mem)
  `(aset1 'mem ,mem ,n ,page))

(defun read-page (offset page)
  (declare (xargs :guard (and (page-p page)
                              (integerp offset) (<= 0 offset)
                              (< offset *num-page-words*))
```

```
                :verify-guards nil))
  (aref1 (page-tag page) (page-array page) offset))

(defun read-mem (addr mem)
  (declare (xargs :guard (and (addr-p addr) (mem-p mem))
                  :verify-guards nil))
  (let ((page (get-page (page-num addr) mem)))
    (if (integerp page)
        0
        (read-page (page-offset addr) page))))

(verify-guards read-page
  :hints (("Goal" :in-theory (enable page-p page-array-p))))

(encapsulate nil
(local
 (defthm page-p-assoc-mem-array
     (implies (and (mem-array-p mem)
                   (integerp pn)
                   (assoc pn mem)
                   (not (integerp (cdr (assoc pn mem)))))
              (page-p (cdr (assoc pn mem))))
   :hints (("Goal" :in-theory (enable assoc mem-array-p)))))

(local
 (defthm integerp-default-in-mem-array
     (implies (mem-p mem)
              (integerp (default 'mem mem)))
   :hints (("Goal" :in-theory (enable mem-p)))))

(defthm page-p-get-page
    (implies (and (mem-p mem)
                  (integerp pn)
                  (not (integerp (get-page pn mem))))
             (page-p (get-page pn mem)))
  :hints (("Goal" :in-theory (enable aref1 mem-p))))
)

(local
 (defthm word-p-assoc-word-array
     (implies (and (word-array-p wa)
                   (integerp pn)
                   (assoc pn wa))
              (word-p (cdr (assoc pn wa))))
   :hints (("Goal" :in-theory (enable word-array-p assoc)))))

(encapsulate nil
(defthm word-p-read-page
    (implies (and (page-p page)
                  (integerp offset))
             (word-p (read-page offset page)))
  :hints (("Goal" :in-theory (enable page-p aref1 page-array-p))))
)

(in-theory (disable read-page read-mem))
(verify-guards read-mem
               :hints (("goal" :in-theory (enable mem-p))))

(defthm word-p-read-mem
    (implies (and (mem-p mem)
                  (addr-p addr))
             (word-p (read-mem addr mem)))
```

```
  :hints (("Goal" :in-theory (enable read-mem)))
  :rule-classes
  ((:rewrite)
   (:type-prescription :corollary
                       (implies (and (mem-p mem) (addr-p addr))
                                (and (integerp (read-mem addr mem))
                                     (>= (read-mem addr mem) 0))))
   (:rewrite :corollary
             (implies (and (mem-p mem) (addr-p addr))
                      (acl2-numberp (read-mem addr mem))))))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Definition of Write-Mem
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Page tag is calculated for fast array accesses.
; The tag for the <n>'th page is given by page<n>.
(defun gen-page-tag (page-num)
  (declare (xargs :guard (and (integerp page-num)
                              (>= page-num 0))
                  :verify-guards nil))
  (pack-intern 'mem
       (coerce (append (coerce (symbol-name 'page) 'list)
                       (explode-nonnegative-integer page-num nil))
               'string)))

(encapsulate nil
(local
 (defthm character-listp-explode-nonnegative-integer-help
     (implies (and (integerp n) (>= n 0)
                   (character-listp x))
              (character-listp (explode-nonnegative-integer n x)))
   :hints (("goal" :in-theory (enable explode-nonnegative-integer
                                      character-listp)))))

(local
 (defthm character-listp-explode-nonnegative-integer
     (implies (and (integerp n) (>= n 0))
              (character-listp (explode-nonnegative-integer n nil)))))

(local
 (defthm true-listp-explode-nonnegative-integer
     (implies (true-listp x)
              (true-listp (explode-nonnegative-integer n x)))))

(verify-guards gen-page-tag
     :hints (("goal" :in-theory (enable U::coerce-string-designator-list
                                        U::STRING-DESIGNATOR-LISTP
                                        character-listp
                                        binary-append))))
)

(defun init-page (page-num mode)
  (declare (xargs :guard (and (integerp page-num) (<= 0 page-num)
                              (integerp mode))
                  :verify-guards nil))
  (let ((name (gen-page-tag page-num)))
    (page name
          mode
          (compress1 name `((:header :name ,name
                                     :dimensions (,*num-page-words*)
```

```
                            :default 0
                            :maximum-length 4096))))))

(verify-guards init-page
 :hints (("Goal" :in-theory (enable array1p alistp
                                    keyword-value-listp
                                    assoc-eq
                                    assoc-keyword
                                    bounded-integer-alistp))))

(defun write-page (val offset page)
  (declare (xargs :guard (and (word-p val)
                              (integerp offset) (<= 0 offset)
                              (< offset *num-page-words*)
                              (page-p page))
                  :verify-guards nil))
  (update-page page
               :array (aset1 (page-tag page) (page-array page) offset val)))

(defun write-mem (val addr mem)
  (declare (xargs :guard (and (word-p val) (addr-p addr) (mem-p mem))
                  :verify-guards nil))
  (let ((page (get-page (page-num addr) mem)))
    (if (integerp page)
        (let ((p (init-page (page-num addr) page)))
          (set-page (write-page val (page-offset addr) p)
                    (page-num addr)
                    mem))
        (set-page (write-page val (page-offset addr) page)
                  (page-num addr)
                  mem))))

(verify-guards write-page
  :hints (("Goal" :in-theory (enable page-p page-array-p))))

(in-theory (disable  write-mem init-page gen-page-tag write-page))

(defthm page-p-init-page
    (implies (integerp mode)
             (page-p (init-page pn mode)))
  :hints (("goal" :in-theory (enable page-p init-page page-array-p
                                     default dimensions
                                     header array1p word-array-p))))

(verify-guards write-mem
  :hints (("Goal" :in-theory (enable mem-p ))))

(local
(defthm word-array-p-compress11
    (implies (and (array1p tag array)
                  (word-array-p array)
                  (integerp i))
             (word-array-p (compress11 tag array i dim default)))
  :hints (("Goal" :in-theory (enable word-array-p)))))

(defthm word-array-p-compress1
    (implies (and (array1p tag array)
                  (word-array-p array))
             (word-array-p (compress1 tag array)))
  :hints (("Goal" :in-theory (enable compress1 word-array-p))))

(defthm word-array-p-aset1
```

```
    (implies (and (array1p tag page-array)
                  (word-array-p page-array)
                  (integerp index)
                  (>= index 0)
                  (> (car (dimensions tag page-array)) index)
                  (word-p val))
             (word-array-p (aset1 tag page-array index val)))
  :hints (("goal" :in-theory (enable aset1 word-array-p ARRAY1P-CONS))))

(defthm page-array-p-aref1
    (implies (and (page-array-p tag page-array)
                  (word-p val)
                  (integerp offset)
                  (>= offset 0)
                  (> *num-page-words* offset))
             (page-array-p tag (aset1 tag page-array offset val)))
  :hints (("Goal" :in-theory (enable page-array-p))))

(defthm page-p-write-page
    (implies (and (word-p val)
                  (integerp offset)
                  (>= offset 0)
                  (> *num-page-words* offset)
                  (page-p page))
             (page-p (write-page val offset page)))
  :hints (("Goal" :in-theory (enable write-page))))

(local
 (defthm valid-integer-assoc-mem-array
     (implies (and (mem-array-p ma)
                   (integerp pn)
                   (assoc pn ma)
                   (integerp (cdr (assoc pn ma)))
                   (not (equal (cdr (assoc pn ma)) 0))
                   (not (equal (cdr (assoc pn ma)) 1)))
              (equal (cdr (assoc pn ma)) 2))
   :hints (("Goal" :in-theory (enable mem-array-p assoc)))))

(local
 (defthm page-p-assoc-mem-array
     (implies (and (mem-array-p ma)
                   (integerp pn)
                   (assoc pn ma)
                   (not (integerp (cdr (assoc pn ma)))))
              (page-p (cdr (assoc pn ma))))
   :hints (("Goal" :in-theory (enable mem-array-p assoc)))))

(local
(defthm mem-array-p-compress11
    (implies (and (array1p tag array)
                  (mem-array-p array)
                  (integerp i))
             (mem-array-p (compress11 tag array i dim default)))
  :hints (("Goal" :in-theory (enable mem-array-p compress1)))))

(defthm mem-array-p-compress1
    (implies (and (array1p tag array)
                  (mem-array-p array))
             (mem-array-p (compress1 tag array)))
  :hints (("Goal" :in-theory (enable mem-array-p compress1))))

(defthm mem-array-p-aset1
```

```
    (implies (and (array1p tag mem-array)
                  (mem-array-p mem-array)
                  (or (page-p page)
                      (equal page *no-access*)
                      (equal page *read-only*)
                      (equal page *read-write*))
                  (integerp pn)
                  (>= pn 0)
                  (> (car (dimensions tag mem-array)) pn))
             (mem-array-p (aset1 tag mem-array pn page)))
  :hints (("Goal" :in-theory (enable mem-array-p aset1))))

(defthm mem-p-write-mem
    (implies (and (word-p val)
                  (addr-p addr)
                  (mem-p mem))
             (mem-p (write-mem val addr mem)))
  :hints (("goal" :in-theory (enable mem-p write-mem))))

(defthm page-num-offset-extensionality
    (implies (and (addr-p addr1)
                  (addr-p addr2)
                  (equal (page-num addr1) (page-num addr2))
                  (equal (page-offset addr1) (page-offset addr2)))
             (equal addr1 addr2))
  :hints (("Goal" :in-theory (enable addr-p page-num page-offset)))
  :rule-classes
  ((:rewrite :corollary
             (implies (and (addr-p addr1)
                           (addr-p addr2)
                           (equal (page-num addr1) (page-num addr2))
                           (not (equal addr1 addr2)))
                      (equal (equal (page-offset addr1) (page-offset addr2))
                             nil)))))

(defthm read-page-init-page
    (implies (integerp offset)
             (equal (read-page offset (init-page pn mode)) 0))
  :hints (("Goal" :in-theory (enable init-page read-page aref1
                                     default header))))

(defthm read-page-write-page
    (implies (and (page-p page)
                  (integerp offset1)
                  (>= offset1 0)
                  (> *num-page-words* offset1)
                  (integerp offset2)
                  (>= offset2 0)
                  (> *num-page-words* offset2))
             (equal (read-page offset1 (write-page val offset2 page))
                    (if (equal offset1 offset2)
                        val
                        (read-page offset1 page))))
  :hints (("Goal" :in-theory (enable page-p read-page write-page
                                     page-array-p))))

(defthm read-mem-write-mem
    (implies (and (addr-p a1)
                  (addr-p a2)
                  (mem-p mem))
             (equal (read-mem a1 (write-mem val a2 mem))
                    (if (equal a1 a2) val (read-mem a1 mem)))))
```

```
  :hints (("Goal" :in-theory (enable read-mem write-mem
                                     mem-p))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Definition of Protection Checks.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun readable-addr-p (ad mem)
  (declare (xargs :guard (and (addr-p ad) (mem-p mem))
                  :verify-guards nil))
  (let ((page (get-page (page-num ad) mem)))
    (if (integerp page)
        (or (equal page *read-only*)
            (equal page *read-write*))
        (or (equal (page-mode page) *read-only*)
            (equal (page-mode page) *read-write*)))))

(verify-guards  readable-addr-p
 :hints (("Goal" :in-theory (enable mem-p))))

(defun readable-addr? (ad mem)
  (declare (xargs :guard (and (addr-p ad) (mem-p mem))))
  (if (readable-addr-p ad mem) 1 0))

(defthm bitp-readable-addr (bitp (readable-addr? ad mem)))

(defun writable-addr-p (ad mem)
  (declare (xargs :guard (and (addr-p ad) (mem-p mem))
                  :verify-guards nil))
  (let ((page (get-page (page-num ad) mem)))
    (if (integerp page)
        (equal page *read-write*)
        (equal (page-mode page) *read-write*))))

(verify-guards  writable-addr-p
 :hints (("Goal" :in-theory (enable mem-p))))

(defun writable-addr? (ad mem)
  (declare (xargs :guard (and (addr-p ad) (mem-p mem))))
  (if (writable-addr-p ad mem) 1 0))

(defthm bitp-writable-addr (bitp (writable-addr? ad mem)))

(defun set-page-mode (mode pn mem)
  (declare (xargs :guard (and (integerp mode)
                              (integerp pn) (<= 0 pn) (< pn *num-pages*)
                              (mem-p mem))
                  :verify-guards nil))
  (let ((page (get-page pn mem)))
    (if (integerp page)
        (set-page mode pn mem)
        (set-page (update-page page :mode mode) pn mem))))

(verify-guards set-page-mode
        :hints (("Goal" :in-theory (enable mem-p))))

(defthm mem-p-set-page-mode
    (implies (and (mem-p mem)
                  (integerp mode)
                  (or (equal mode *no-access*)
                      (equal mode *read-only*)
                      (equal mode *read-write*))
                  (integerp pn) (<= 0 pn) (< pn *num-pages*))
```

```
            (mem-p (set-page-mode mode pn mem)))
  :hints (("Goal" :in-theory (enable mem-p))))

(defthm page-mode-init-page
    (equal (page-mode (init-page page-num mode)) mode)
  :hints (("Goal" :in-theory (enable init-page))))

(defthm page-mode-write-page
    (equal (page-mode (write-page val offset page))
           (page-mode page))
  :hints (("Goal" :in-theory (enable write-page))))

(defthm readable-addr-p-set-page-mode
    (implies (and (integerp mode)
                  (addr-p addr)
                  (integerp pn1) (<= 0 pn1) (< pn1 *num-pages*)
                  (mem-p mem))
             (equal (readable-addr-p addr (set-page-mode mode pn1 mem))
                    (if (equal (page-num addr) pn1)
                        (or (equal mode *read-only*) (equal mode *read-write*))
                        (readable-addr-p addr mem))))
  :hints (("Goal" :in-theory (enable set-page-mode mem-p
                                     readable-addr-p))))

(defthm readable-addr-set-page-mode
    (implies (and (integerp mode)
                  (addr-p addr)
                  (integerp pn1) (<= 0 pn1) (< pn1 *num-pages*)
                  (mem-p mem))
             (equal (readable-addr? addr (set-page-mode mode pn1 mem))
                    (if (equal (page-num addr) pn1)
                        (if (or (equal mode *read-only*)
                                (equal mode *read-write*))
                            1 0)
                        (readable-addr? addr mem))))
  :hints (("Goal" :in-theory (e/d (readable-addr?) (readable-addr-p
                                                    SET-PAGE-MODE)))))

(defthm writable-addr-p-set-page-mode
    (implies (and (integerp mode)
                  (addr-p addr)
                  (integerp pn1) (<= 0 pn1) (< pn1 *num-pages*)
                  (mem-p mem))
             (equal (writable-addr-p addr (set-page-mode mode pn1 mem))
                    (if (equal (page-num addr) pn1)
                        (equal mode *read-write*)
                        (writable-addr-p addr mem))))
  :hints (("Goal" :in-theory (enable set-page-mode mem-p
                                     writable-addr-p))))

(defthm writable-addr-set-page-mode
    (implies (and (integerp mode)
                  (addr-p addr)
                  (integerp pn1) (<= 0 pn1) (< pn1 *num-pages*)
                  (mem-p mem))
             (equal (writable-addr? addr (set-page-mode mode pn1 mem))
                    (if (equal (page-num addr) pn1)
                        (if (equal mode *read-write*) 1 0)
                        (writable-addr? addr mem))))
  :hints (("Goal" :in-theory (e/d (writable-addr?)
                                  (writable-addr-p SET-PAGE-MODE)))))
```

```
(defthm readable-addr-p-write-mem
    (implies (and (addr-p addr) (addr-p addr2) (word-p val) (mem-p mem))
             (equal (readable-addr-p addr (write-mem val addr2 mem))
                    (readable-addr-p addr mem)))
  :hints (("Goal" :in-theory (enable readable-addr-p write-mem mem-p))))

(defthm readable-addr-write-mem
    (implies (and (addr-p addr) (addr-p addr2) (word-p val) (mem-p mem))
             (equal (readable-addr? addr (write-mem val addr2 mem))
                    (readable-addr? addr mem)))
  :hints (("Goal" :in-theory (enable readable-addr?))))

(defthm writable-addr-p-write-mem
    (implies (and (addr-p addr) (addr-p addr2) (word-p val) (mem-p mem))
             (equal (writable-addr-p addr (write-mem val addr2 mem))
                    (writable-addr-p addr mem)))
  :hints (("Goal" :in-theory (enable writable-addr-p write-mem mem-p))))

(defthm writable-addr-write-mem
    (implies (and (addr-p addr) (addr-p addr2) (word-p val) (mem-p mem))
             (equal (writable-addr? addr (write-mem val addr2 mem))
                    (writable-addr? addr mem)))
  :hints (("Goal" :in-theory (enable writable-addr?))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Initialize Memory
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defconst *init-mem*
    (compress1 'mem (list `(:header :name mem
                            :dimensions (,*num-pages*)
                            :default ,*no-access*
                            :maximum-length 2048))))

(defthm mem-p-init-mem
    (mem-p *init-mem*))

(deflist mem-alist-p (l)
  (declare (xargs :guard t))
  (lambda (l) (and (consp l) (addr-p (car l)) (word-p (cdr l)))))

(defun load-mem-alist (alist mem)
  (declare (xargs :guard (and (mem-alist-p alist) (mem-p mem))))
  (if (endp alist)
      mem
      (load-mem-alist (cdr alist) (write-mem (cdar alist) (caar alist) mem))))

(defthm mem-p-load-mem-alist
    (implies (and (mem-alist-p alist)
                  (mem-p mem))
             (mem-p (load-mem-alist alist mem))))

(in-theory (disable page-p page-array-p word-array-p mem-p))
(in-theory (disable read-reg))
(in-theory (disable write-reg))
(in-theory (disable read-mem))
(in-theory (disable write-mem))
(in-theory (disable writable-addr-p readable-addr-p
                    writable-addr? readable-addr? set-page-mode))

(defword* word-layout ((opcode 4 12)
                       (rc *rname-size* 8)
                       (ra *rname-size* 4)
```

```
                      (rb *rname-size* 0)
                      (im *immediate-size* 0))
  :conc-name ||)

(defthm opcd-p-opcode
    (opcd-p (opcode inst))
  :hints (("Goal" :in-theory (enable opcd-p))))

(defthm rname-p-rc
    (rname-p (rc inst))
  :hints (("Goal" :in-theory (enable rname-p))))

(defthm rname-p-ra
    (rname-p (ra inst))
  :hints (("Goal" :in-theory (enable rname-p))))

(defthm rname-p-rb
    (rname-p (rb inst))
  :hints (("Goal" :in-theory (enable rname-p))))

(defthm immediate-p-immediate-field
    (immediate-p (im inst))
  :hints (("Goal" :in-theory (enable immediate-p))))

(defthm word-p-immediate-field
    (word-p (im inst))
  :hints (("Goal" :in-theory (enable word-p))))

(defthm word-im
    (equal (word (im i)) (im i))
  :hints (("goal" :in-theory (enable word))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Definition of Special registers
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defconst *num-sregs* 2)

(defstructure SRF
  (su (:assert (bitp su) :rewrite ))
  (sr0 (:assert (word-p sr0) :rewrite))
  (sr1 (:assert (word-p sr1) :rewrite))
  (:options :guards))

(defun srname-p (rname)
  (declare (xargs :guard t))
  (and (rname-p rname) (< rname *num-sregs*)))

(defthm srname-p-type
    (implies (srname-p rname)
             (and (integerp rname)
                  (>= rname 0)
                  (< rname *num-sregs*)))
  :hints (("Goal" :in-theory (enable rname-p srname-p)))
  :rule-classes :forward-chaining)

(defun read-sreg (r SRF)
  (declare (xargs :guard (and (rname-p r) (SRF-p SRF))))
  (if (equal r 0) (SRF-sr0 SRF)
      (if (equal r 1) (SRF-sr1 SRF)
          0)))

(defun write-sreg (val r SRF)
```

```
  (declare (xargs :guard (and (word-p val) (rname-p r) (SRF-p SRF))))
  (if (equal r 0) (SRF (SRF-su SRF) val (SRF-sr1 SRF))
      (if (equal r 1) (SRF (SRF-su SRF) (SRF-sr0 SRF) val)
          SRF)))

(defthm word-p-read-sreg
    (implies (and (rname-p r) (SRF-p SRF))
             (word-p (read-sreg r SRF))))

(defthm numberp-read-sreg
    (implies (and (SRF-p SRF)
                  (rname-p rname))
             (and (integerp (read-sreg rname SRF))
                  (acl2-numberp (read-sreg rname SRF))))
  :hints (("Goal" :in-theory (enable rname-p SRF-p fixlen-word-listp)))
  :rule-classes
  ((:type-prescription) (:rewrite)))

(defthm SRF-p-write-sreg
    (implies (and (word-p val) (rname-p r) (SRF-p SRF))
             (SRF-p (write-sreg val r SRF))))

(defthm read-sreg-write-sreg
    (implies (and (srname-p r1)
                  (srname-p r2)
                  (SRF-p SRF))
             (equal (read-sreg r1 (write-sreg val r2 SRF))
                    (if (equal r1 r2) val (read-sreg r1 SRF))))
  :hints (("Goal" :in-theory (enable read-sreg write-sreg SRF-p
                                     srname-p))))

(in-theory (disable read-sreg write-sreg srname-p))
```

D.2.2 ISA-def.lisp

This file contains the definition of the FM9801 ISA.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ISA-def.lisp:
; Author  Jun Sawada, University of Texas at Austin
;
;  This file includes the definitions of our ISA.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; The definition ISA behavior is given by (ISA-step ISA intr), which
; returns the state of ISA after executing one instructions.
; (ISA-stepn ISA intr-lst n) returns the state of ISA after n instructions
; are executed.
;
; This ISA implements following exceptions.
;  External Interrupt
;  Fetch Error
;  Illegal instruction
;  Data Access Error
;
; When an exception happens, following actions are taken by the machine.
;   su <--- 0  (Set Supervisor mode)
;   sr0 <-- The restarting address after exception handling
;   sr1 <-- su
;   pc  <-- Exception Vectors
;   RF <-- All writes to the registers take place before the exception
```

```
;            handling takes place.
;   mem <-- All writes to the memory by the preceding instructions must
;           complete.  If another error happens during the completion,
;           the first instruction in program order is considered to
;           have caused the memory error.
;
; External Interrupt
;  The machine will complete all issued instructions, but does not
;  issue any new instructions.  SR0 will point to the next instruction
;  to be executed after the exception handling.  Exception vector is
;  0x30
;
; Fetch Error
;  If the instruction fetch address is not readable, a fetch error
;  will be raised.  SR0 will point to the instruction address which
;  the processor failed to access.  Exception Vector is 0x10.  Note
;  that speculative fetch may cause a fake fetch error, but the
;  machine should not go into the exception handling cycle.
;
; Data Access Error
;  If the processor fails to load or store data on the memory, an data
;  access error occurs and the exception is raised. Sr0 points to the
;  instruction that caused the data access error.  Exception vector is
;  0x20.
;
; Illegal instruction
;  If the processor tries to execute an undefined instruction, or it
;  tries to execute a privileged instruction from the user mode, an
;  illegal instruction exception is raised and the processor goes into
;  the exception handling.  SR0 points to the next instruction after
;  the Illegal instruction.  The exception vector is 0x0
;
; Memory protection issues on Supervisor and User mode.
;  Memory protection is effective only in the user mode.  In the
;  supervisor mode, the processor can access any address.  Memory
;  address translation does not exist in our model.
;
; Added instructions
;  In order to control the exception handling, we add a few more instructions.
; RFEH (Return From Exception Handling) privileged instruction
;     su <- SR1 & 0x1
;     pc <- Sr0
; MFSR rc (Move From Special Register) Privileged instruction
;     rc <- SR0
; MTSR rc (Move To Special Register) Privileged instruction
;     SR0 <- rc
; SPM rc,ra+rb (Set page mode) (proposed, not implemented)
;     The access mode of the page containing ra+rb address is
;     set to rc.  If rc does not contain 0, 1 or 2, the page mode is
;     set to no-access.
;
(in-package "ACL2")

(include-book "utils")
(include-book "basic-def")
(include-book "b-ops-aux")

;; Beginning of the definition of the Instruction-Set Architecture.
(deflabel begin-ISA-def)

(deflabel begin-ISA-state-def)
```

```
; An ISA state consists of a program counter, register file
; special register file and memory.
(defstructure ISA-state
  (pc (:assert (addr-p pc) :rewrite (:rewrite (Integerp pc))))
  (RF (:assert (RF-p RF) :rewrite))
  (SRF (:assert (SRF-p SRF) :rewrite))
  (mem (:assert (mem-p mem) :rewrite))
  (:options :guards  (:conc-name ISA-)))

; An ISA input contains a flag for external interrupt.
(defstructure ISA-input
  (exint (:assert (bitp exint) :rewrite))
  (:options :guards))

(deflist ISA-input-listp (l)
  (declare (xargs :guard t))
  ISA-input-p)

(deflabel end-ISA-state-def)

(defun read-error? (addr mem su)
  (declare (xargs :guard (and (addr-p addr) (mem-p mem) (bitp su))))
  (b-nor su (readable-addr? addr mem)))

(defthm bitp-read-error (bitp (read-error? addr mem su)))

(defun write-error? (addr mem su)
  (declare (xargs :guard (and (addr-p addr) (mem-p mem) (bitp su))))
  (b-nor su (writable-addr? addr mem)))

(defthm bitp-write-error (bitp (write-error? addr mem su)))

(deflabel begin-ISA-step-functions)

(defun supervisor-mode? (ISA)
  (declare (xargs :guard (ISA-state-p ISA)))
  (SRF-su (ISA-SRF ISA)))

(defthm bitp-supervisor-mode
    (implies (ISA-state-p ISA) (bitp (supervisor-mode? ISA))))

;; Definitions of states after jumping to exception handling states.
(defun ISA-fetch-error (ISA)
  (declare (xargs :guard (ISA-state-p ISA)))
  (ISA-state #x10
             (ISA-RF ISA)
             (SRF 1 (word (ISA-pc ISA)) (word (SRF-su (ISA-SRF ISA))))
             (ISA-mem ISA)))

(defun ISA-data-accs-error (ISA)
  (declare (xargs :guard (ISA-state-p ISA)))
  (ISA-state #x20
             (ISA-RF ISA)
             (SRF 1 (word (ISA-pc ISA)) (word (SRF-su (ISA-SRF ISA))))
             (ISA-mem ISA)))

(defun ISA-illegal-inst (ISA)
  (declare (xargs :guard (ISA-state-p ISA)))
  (ISA-state #x0
             (ISA-RF ISA)
             (SRF 1 (word (1+ (ISA-pc ISA))) (word (SRF-su (ISA-SRF ISA))))
             (ISA-mem ISA)))
```

```
(defun ISA-external-intr (ISA)
  (declare (xargs :guard (ISA-state-p ISA)))
  (ISA-state #x30
             (ISA-RF ISA)
             (SRF 1 (word (ISA-pc ISA)) (word (SRF-su (ISA-SRF ISA))))
             (ISA-mem ISA)))

;; In the following definitions, rc ra and rb represent the field
;; value of the current instruction, and ISA represents the current
;; state of the machine.
(defun ISA-add (rc ra rb ISA)
  (declare (xargs :guard (and (rname-p rc) (rname-p ra) (rname-p rb)
                              (ISA-state-p ISA))))
  (let* ((pc (ISA-pc ISA))
         (RF (ISA-RF ISA))
         (val (word (+ (read-reg ra RF) (read-reg rb RF)))))
    (ISA-state (addr (1+ pc))
               (write-reg val rc RF)
               (ISA-SRF ISA)
               (ISA-mem ISA))))

(defun ISA-mul (rc ra rb ISA)
  (declare (xargs :guard (and (rname-p rc) (rname-p ra) (rname-p rb)
                              (ISA-state-p ISA))))
  (let* ((pc (ISA-pc ISA))
         (RF (ISA-RF ISA))
         (val (word (* (read-reg ra RF) (read-reg rb RF)))))
    (ISA-state (addr (1+ pc))
               (write-reg val rc RF)
               (ISA-SRF ISA)
               (ISA-mem ISA))))

(defun ISA-br (rc im ISA)
  (declare (xargs :guard (and (rname-p rc) (immediate-p im)
                              (ISA-state-p ISA))))
  (let ((RF (ISA-RF ISA))
        (pc (ISA-pc ISA)))
    (ISA-state (if (equal (read-reg rc RF) 0)
                   (addr (+ (logextu *addr-size* *immediate-size* im) pc))
                   (addr (1+ pc)))
               (ISA-RF ISA)
               (ISA-SRF ISA)
               (ISA-mem ISA))))

(defun ISA-ld (rc ra rb ISA)
  (declare (xargs :guard (and (rname-p rc) (rname-p ra) (rname-p rb)
                              (ISA-state-p ISA))))
  (let* ((pc (ISA-pc ISA))
         (RF (ISA-RF ISA))
         (SRF (ISA-SRF ISA))
         (mem (ISA-mem ISA))
         (ad (addr (+ (read-reg ra RF) (read-reg rb RF))))
         (val (read-mem ad mem)))
    (b-if (read-error? ad mem (SRF-su SRF))
          (ISA-data-accs-error ISA)
          (ISA-state (addr (1+ pc))
                     (write-reg val rc RF)
                     (ISA-SRF ISA)
                     (ISA-mem ISA)))))

(defun ISA-ldi (rc im ISA)
```

```
  (declare (xargs :guard (and (rname-p rc) (immediate-p im)
                              (ISA-state-p ISA))))
  (let* ((pc (ISA-pc ISA))
         (RF (ISA-RF ISA))
         (SRF (ISA-SRF ISA))
         (mem (ISA-mem ISA))
         (ad (addr im))
         (val (read-mem ad mem)))
    (b-if (read-error? ad mem (SRF-su SRF))
          (ISA-data-accs-error ISA)
          (ISA-state (addr (1+ pc))
                     (write-reg val rc RF)
                     (ISA-SRF ISA)
                     (ISA-mem ISA)))))

(defun ISA-st (rc ra rb ISA)
  (declare (xargs :guard (and (rname-p rc) (rname-p ra) (rname-p rb)
                              (ISA-state-p ISA))))
  (let* ((pc (ISA-pc ISA))
         (RF (ISA-RF ISA))
         (SRF (ISA-SRF ISA))
         (mem (ISA-mem ISA))
         (ad (addr (+ (read-reg ra RF) (read-reg rb RF))))
         (val (read-reg rc RF)))
    (b-if (write-error? ad mem (SRF-su SRF))
          (ISA-data-accs-error ISA)
          (ISA-state (addr (1+ pc))
                     (ISA-RF ISA)
                     (ISA-SRF ISA)
                     (write-mem val ad mem)))))

(defun ISA-sti (rc im ISA)
  (declare (xargs :guard (and (rname-p rc) (immediate-p im)
                              (ISA-state-p ISA))))
  (let* ((pc (ISA-pc ISA))
         (RF (ISA-RF ISA))
         (SRF (ISA-SRF ISA))
         (mem (ISA-mem ISA))
         (ad (addr im))
         (val (read-reg rc RF)))
    (b-if (write-error? ad mem (SRF-su SRF))
          (ISA-data-accs-error ISA)
          (ISA-state (addr (1+ pc))
                     (ISA-RF ISA)
                     (ISA-SRF ISA)
                     (write-mem val ad mem)))))

(defun ISA-rfeh (ISA)
  (declare (xargs :guard (ISA-state-p ISA)))
  (b-if (supervisor-mode? ISA)
        (let* ((SRF (ISA-SRF ISA))
               (sr0 (read-sreg 0 SRF))
               (sr1 (read-sreg 1 SRF)))
          (ISA-state (addr sr0)
                     (ISA-RF ISA)
                     (SRF (logcar sr1) sr0 sr1)
                     (ISA-mem ISA)))
        (ISA-illegal-inst ISA)))

(defun ISA-mfsr (rc ra ISA)
  (declare (xargs :guard (and (rname-p rc) (rname-p ra) (ISA-state-p ISA))))
  (cond ((zbp (supervisor-mode? ISA))
```

```
         (ISA-illegal-inst ISA))
        ((or (equal ra 0) (equal ra 1))
         (let* ((pc (ISA-pc ISA))
                (RF (ISA-RF ISA))
                (SRF (ISA-SRF ISA))
                (val (read-sreg ra SRF)))
           (ISA-state (addr (1+ pc))
                      (write-reg val rc RF)
                      (ISA-SRF ISA)
                      (ISA-mem ISA))))
        (t (ISA-illegal-inst ISA))))

(defun ISA-mtsr (rc ra ISA)
  (declare (xargs :guard (and (rname-p rc) (rname-p ra) (ISA-state-p ISA))))
  (cond ((zbp (supervisor-mode? ISA))
         (ISA-illegal-inst ISA))
        ((or (equal ra 0) (equal ra 1))
         (let* ((pc (ISA-pc ISA))
                (RF (ISA-RF ISA))
                (SRF (ISA-SRF ISA))
                (val (read-reg rc RF)))
           (ISA-state (addr (1+ pc))
                      (ISA-RF ISA)
                      (write-sreg val ra SRF)
                      (ISA-mem ISA))))
        (t (ISA-illegal-inst ISA))))

(defun ISA-sync (ISA)
  (declare (xargs :guard (ISA-state-p ISA)))
  (ISA-state (addr (1+ (ISA-pc ISA)))
             (ISA-RF ISA)
             (ISA-SRF ISA)
             (ISA-mem ISA)))

(deflabel end-ISA-step-functions)

; ISA-step takes the current state, ISA, and the interrupt signal, intr,
; and returns the next state.
(defun ISA-step (ISA intr)
  "ISA represents the current state."
  (declare (xargs :guard (and (ISA-state-p ISA) (ISA-input-p intr))))
  (b-if (ISA-input-exint intr)
        (ISA-external-intr ISA)
  ; otherwise
  (b-if (read-error? (ISA-pc ISA) (ISA-mem ISA) (SRF-su (ISA-SRF ISA)))
        (ISA-fetch-error ISA)
        (let ((inst (read-mem (ISA-pc ISA) (ISA-mem ISA))))
          (let ((op (opcode inst))
                (rc (rc inst))
                (ra (ra inst))
                (rb (rb inst))
                (im (im inst)))
            (cond ((equal op 0)         ; add
                   (ISA-add rc ra rb ISA))
                  ((equal op 1)         ; multiply
                   (ISA-mul rc ra rb ISA))
                  ((equal op 2)         ; conditional branch
                   (ISA-br rc im ISA))
                  ((equal op 3)         ; load
                   (ISA-ld rc ra rb ISA))
                  ((equal op 6)         ; load from an immediate address
                   (ISA-ldi rc im ISA))
```

```
                 ((equal op 4)         ; store
                  (ISA-st rc ra rb ISA))
                 ((equal op 7)         ; store at an immediate address
                  (ISA-sti rc im ISA))
                 ((equal op 5)         ; sync
                  (ISA-sync ISA))
                 ((equal op 8) ; RFEH
                  (ISA-rfeh ISA))
                 ((equal op 9) ; MFSR
                  (ISA-mfsr rc ra ISA))
                 ((equal op 10) ; MTSR
                  (ISA-mtsr rc ra ISA))
                 (t (ISA-illegal-inst ISA))))))))

; Runs the ISA machine n-steps.
; Argument intr-lst is the list of interrupt signals.
(defun ISA-stepn (ISA intr-lst n)
  (declare (xargs :guard (and (ISA-state-p ISA) (integerp n) (>= n 0)
                              (ISA-input-listp intr-lst)
                              (<= n (len intr-lst)))
                  :verify-guards nil))
  (if (zp n)
      ISA
      (ISA-stepn (ISA-step ISA (car intr-lst)) (cdr intr-lst) (1- n))))

(verify-guards ISA-stepn)

(defthm ISA-state-p-ISA-fetch-error
    (implies (ISA-state-p ISA)
             (ISA-state-p (ISA-fetch-error ISA))))

(defthm ISA-state-p-ISA-data-accs-error
    (implies (ISA-state-p ISA)
             (ISA-state-p (ISA-data-accs-error ISA))))

(defthm ISA-state-p-ISA-illegal-inst
    (implies (ISA-state-p ISA)
             (ISA-state-p (ISA-illegal-inst ISA))))

(defthm ISA-state-p-ISA-external-intr
    (implies (ISA-state-p ISA)
             (ISA-state-p (ISA-external-intr ISA))))

(defthm ISA-state-p-ISA-add
    (implies (and (ISA-state-p ISA)
                  (rname-p rc))
             (ISA-state-p (ISA-add rc ra rb ISA))))

(defthm ISA-state-p-ISA-mul
    (implies (and (ISA-state-p ISA)
                  (rname-p rc))
             (ISA-state-p (ISA-mul rc ra rb ISA))))

(defthm ISA-state-p-ISA-br
    (implies (ISA-state-p ISA)
             (ISA-state-p (ISA-br rc im ISA))))

(defthm ISA-state-p-ISA-ld
    (implies (and (ISA-state-p ISA)
                  (rname-p rc))
             (ISA-state-p (ISA-ld rc ra rb ISA))))
```

```
(defthm ISA-state-p-ISA-ldi
    (implies (and (ISA-state-p ISA)
                  (rname-p rc))
             (ISA-state-p (ISA-ldi rc im ISA))))

(defthm ISA-state-p-ISA-st
    (implies (and (ISA-state-p ISA)
                  (rname-p rc))
             (ISA-state-p (ISA-st rc ra rb ISA))))

(defthm ISA-state-p-ISA-sti
    (implies (and (ISA-state-p ISA) (rname-p rc))
             (ISA-state-p (ISA-sti rc im ISA))))

(defthm ISA-state-p-ISA-rfeh
    (implies (ISA-state-p ISA)
             (ISA-state-p (ISA-rfeh ISA))))

(defthm ISA-state-p-ISA-mfsr
    (implies (and (ISA-state-p ISA)
                  (rname-p rc))
             (ISA-state-p (ISA-mfsr rc ra ISA))))

(defthm ISA-state-p-ISA-mtsr
    (implies (and (ISA-state-p ISA)
                  (rname-p rc))
             (ISA-state-p (ISA-mtsr rc ra ISA))))

(defthm ISA-state-p-ISA-sync
    (implies (ISA-state-p ISA)
             (ISA-state-p (ISA-sync ISA))))

(defthm ISA-state-p-ISA-step
    (implies (ISA-state-p ISA)
             (ISA-state-p (ISA-step ISA intr))))

(in-theory (disable ISA-step))

(defthm ISA-state-p-ISA-stepn
    (implies (ISA-state-p ISA)
             (ISA-state-p (ISA-stepn ISA intr-lst n))))

(deflabel end-ISA-def)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Definition of No-Self-Modifying Code
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deflabel begin-ISA-functions)

(defun store-inst-p (inst)
  (declare (xargs :guard (word-p inst)))
  (or (equal (opcode inst) 7) (equal (opcode inst) 4)))

(defun ISA-store-inst-p (ISA)
  (declare (xargs :guard (ISA-state-p ISA)))
  (store-inst-p (read-mem (ISA-pc ISA) (ISA-mem ISA))))

(defun ISA-store-addr (ISA)
  (declare (xargs :guard (ISA-state-p ISA)))
  (let ((inst (read-mem (ISA-pc ISA) (ISA-mem ISA)))
        (RF (ISA-RF ISA)))
    (cond ((equal (opcode inst) 7)
```

```
          (addr (im inst)))
         ((equal (opcode inst) 4)
          (addr (+ (read-reg (ra inst) RF)
                   (read-reg (rb inst) RF))))
         (t 0))))

(defun ISA-fetch-error-p (ISA)
  (declare (xargs :guard (ISA-state-p ISA)))
  (b1p (read-error? (ISA-pc ISA) (ISA-mem ISA) (SRF-su (ISA-SRF ISA)))))

(defun ISA-decode-error-p (ISA)
  (declare (xargs :guard (ISA-state-p ISA)))
  (let ((opcd (opcode (read-mem (ISA-pc ISA) (ISA-mem ISA))))
        (ra (ra (read-mem (ISA-pc ISA) (ISA-mem ISA))))
        (su (SRF-su (ISA-SRF ISA))))
    (not (or (equal opcd 0)
             (equal opcd 1)
             (equal opcd 2)
             (equal opcd 3)
             (equal opcd 4)
             (equal opcd 5)
             (equal opcd 6)
             (equal opcd 7)
             (and (equal opcd 8) (b1p su))
             (and (equal opcd 9) (b1p su) (or (equal ra 0) (equal ra 1)))
             (and (equal opcd 10) (b1p su) (or (equal ra 0) (equal ra 1)))))))

(defun ISA-load-access-error-p (ISA)
  (declare (xargs :guard (ISA-state-p ISA)))
  (let ((inst (read-mem (ISA-pc ISA) (ISA-mem ISA)))
        (su (SRF-su (ISA-SRF ISA)))
        (mem (ISA-mem ISA))
        (RF (ISA-RF ISA)))
    (if (equal (opcode inst) 6)
        (b1p (read-error? (addr (im inst)) mem su))
        (if (equal (opcode inst) 3)
            (b1p (read-error? (addr (+ (read-reg (ra inst) RF)
                                       (read-reg (rb inst) RF)))
                              mem su))
            nil))))

(defun ISA-store-access-error-p (ISA)
  (declare (xargs :guard (ISA-state-p ISA)))
  (let ((inst (read-mem (ISA-pc ISA) (ISA-mem ISA)))
        (su (SRF-su (ISA-SRF ISA)))
        (mem (ISA-mem ISA))
        (RF (ISA-RF ISA)))
    (if (equal (opcode inst) 7)
        (b1p (write-error? (addr (im inst)) mem su))
        (if (equal (opcode inst) 4)
            (b1p (write-error? (addr (+ (read-reg (ra inst) RF)
                                        (read-reg (rb inst) RF)))
                               mem su))
            nil))))

(defun ISA-data-access-error-p (ISA)
  (declare (xargs :guard (ISA-state-p ISA)))
  (or (ISA-load-access-error-p ISA) (ISA-store-access-error-p ISA)))

(defun ISA-excpt-p (ISA)
  (declare (xargs :guard (ISA-state-p ISA)))
  (or (ISA-fetch-error-p ISA)
```

```
      (ISA-decode-error-p ISA)
      (ISA-data-access-error-p ISA)))

; If ISA state transition (ISA-step ISA intr) fetches an instruction
; from addr, (ISA-fetches-from addr ISA intr) returns non-nil.
(defun ISA-fetches-from (addr ISA intr)
  (declare (xargs :guard (and (addr-p addr) (ISA-state-p ISA)
                              (ISA-input-p intr))))
  (and (equal addr (ISA-pc ISA))
       (not (b1p (ISA-input-exint intr)))))

;; If ISA state transition (ISA-stepn ISA intr n) fetches an instruction
; from addr, (ISA-stepn-fetches-from addr ISA intr-lst n) returns non-nil.
(defun ISA-stepn-fetches-from (addr ISA intr-lst n)
  (declare (xargs :guard (and (addr-p addr) (ISA-state-p ISA)
                              (integerp n) (<= 0 n)
                              (ISA-input-listp intr-lst)
                              (<= n (len intr-lst)))
                  :measure (nfix n)))
  (if (zp n)
      nil
      (or (ISA-fetches-from addr ISA (car intr-lst))
          (ISA-stepn-fetches-from addr (ISA-step ISA (car intr-lst))
                                  (cdr intr-lst) (1- n)))))

(defun ISA-self-modify-p (ISA intr-lst n)
  (declare (xargs :guard (and (ISA-state-p ISA) (integerp n) (<= 0 n)
                              (ISA-input-listp intr-lst)
                              (<= n (len intr-lst)))
                  :measure (nfix n)))
  (if (zp n)
      nil
      (or (and (ISA-store-inst-p ISA)
               (not (ISA-excpt-p ISA))
               (not (b1p (ISA-input-exint (car intr-lst))))
               (ISA-stepn-fetches-from (ISA-store-addr ISA)
                                       (ISA-step ISA (car intr-lst))
                                       (cdr intr-lst)
                                       (1- n)))
          (ISA-self-modify-p (ISA-step ISA (car intr-lst))
                             (cdr intr-lst)
                             (1- n)))))

(defun ISA-writes-at (addr ISA intr)
  (declare (xargs :guard (and (addr-p addr) (ISA-state-p ISA)
                              (ISA-input-p intr))))
  (and (ISA-store-inst-p ISA)
       (not (ISA-excpt-p ISA))
       (not (b1p (ISA-input-exint intr)))
       (equal (ISA-store-addr ISA) addr)))

(defun ISA-stepn-writes-at (addr ISA intr-lst n)
  (declare (xargs :guard (and (addr-p addr) (ISA-state-p ISA)
                              (integerp n) (<= 0 n)
                              (ISA-input-listp intr-lst)
                              (<= n (len intr-lst)))
                  :measure (nfix n)))
  (if (zp n)
      nil
      (or (ISA-writes-at addr ISA (car intr-lst))
          (ISA-stepn-writes-at addr (ISA-step ISA (car intr-lst))
                               (cdr intr-lst) (1- n)))))
```

```
(deflabel end-ISA-functions)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ISA theories
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deftheory ISA-def
    (set-difference-theories (function-theory 'end-ISA-def)
                             (function-theory 'begin-ISA-def)))

(deftheory ISA-state-def
    (set-difference-theories (function-theory 'end-ISA-state-def)
                             (function-theory 'begin-ISA-state-def)))

(deftheory ISA-step-functions
  (definition-theory
     (set-difference-theories (universal-theory 'end-ISA-step-functions)
                              (universal-theory 'begin-ISA-step-functions))))
(in-theory (disable ISA-step-functions))

(deftheory ISA-functions
    (set-difference-theories (function-theory 'end-ISA-functions)
                             (function-theory 'begin-ISA-functions)))

(deftheory ISA-non-rec-functions
    (non-rec-functions (theory 'ISA-functions) world))

(in-theory (disable ISA-def))
(in-theory (disable ISA-non-rec-functions))

#|
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Proof of an Equivalent Definition of ISA-self-modify-p
;  This is irrelevant to the main proof.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; We prove that
; (ISA-fetches-from addr ISA intr-lst n)
; <=> (exists i j addr (and (< i j) (< j n)
;                           (ISA-writes-at addr ISA-i intr-i)
;                           (ISA-fetches-from addr ISA-j intr-j))
; We prove this lemma by showing two lemmas ISA-fetches-from* and
; ISA-fetches-from**. ISA-fetches-from* shows the right arrow, and
; ISA-fetches-from** shows the left arrow.
;
; We show the existence of i, j, and addr with the witness functions i, j,
; and write-addr.
(defun j-i-1 (addr ISA intr-lst n)
  (declare (xargs :measure (nfix n)))
  (if (zp n)
      0
      (if (ISA-fetches-from addr ISA (car intr-lst))
          0
          (1+ (j-i-1 addr (ISA-step ISA (car intr-lst))
                     (cdr intr-lst) (1- n))))))

(defun i (ISA intr-lst n)
  (declare (xargs :measure (nfix n)))
  (if (zp n)
      0
      (if (and (ISA-store-inst-p ISA)
               (not (ISA-excpt-p ISA))
               (not (b1p (ISA-input-exint (car intr-lst))))
               (ISA-stepn-fetches-from (ISA-store-addr ISA)
```

```
                                    (ISA-step ISA (car intr-lst))
                                    (cdr intr-lst)
                                    (1- n)))
          0
          (1+ (i (ISA-step ISA (car intr-lst)) (cdr intr-lst) (1- n))))))

(defun j (ISA intr-lst n)
  (declare (xargs :measure (nfix n)))
  (if (zp n)
      0
      (if (and (ISA-store-inst-p ISA)
               (not (ISA-excpt-p ISA))
               (not (b1p (ISA-input-exint (car intr-lst))))
               (ISA-stepn-fetches-from (ISA-store-addr ISA)
                                    (ISA-step ISA (car intr-lst))
                                    (cdr intr-lst)
                                    (1- n)))
          (1+ (j-i-1 (ISA-store-addr ISA)
                     (ISA-step ISA (car intr-lst)) (cdr intr-lst) (1- n)))
          (1+ (j (ISA-step ISA (car intr-lst)) (cdr intr-lst) (1- n))))))

(defun write-addr (ISA intr n)
  (ISA-store-addr (ISA-stepn ISA intr (i ISA intr n))))

(defthm fetch-from-at-the-cycle
    (let ((ISA-i (ISA-stepn ISA-0 intr-lst (j-i-1 addr ISA-0 intr-lst n)))
          (intr-i (nth (j-i-1 addr ISA-0 intr-lst n) intr-lst)))
      (implies (ISA-stepn-fetches-from addr ISA-0 intr-lst n)
               (ISA-fetches-from addr ISA-i intr-i)))
  :hints (("goal" :in-theory (enable ISA-stepn))))

(defthm i-is-right
    (let ((i (i ISA-0 intr-lst n))
          (addr (write-addr ISA-0 intr-lst n)))
      (let ((ISA-i (ISA-stepn ISA-0 intr-lst i))
            (intr-i (nth i intr-lst)))
        (implies (ISA-self-modify-p ISA-0 intr-lst n)
                 (ISA-writes-at addr ISA-i intr-i))))
  :hints (("goal" :in-theory (enable ISA-stepn ISA-WRITES-AT))))

(defthm j-is-right
    (let ((j (j ISA-0 intr-lst n))
          (addr (write-addr ISA-0 intr-lst n)))
      (let ((ISA-j (ISA-stepn ISA-0 intr-lst j))
            (intr-j (nth j intr-lst)))
        (implies (ISA-self-modify-p ISA-0 intr-lst n)
                 (ISA-fetches-from addr ISA-j intr-j))))
  :hints (("goal" :in-theory (enable ISA-stepn ISA-WRITES-AT))))

(in-theory (disable write-addr))

(defthm i-<-j
    (implies (ISA-self-modify-p ISA-0 intr-lst n)
             (< (i ISA-0 intr-lst n) (j ISA-0 intr-lst n))))

(defthm j-i-1-<-n
    (implies (ISA-stepn-fetches-from addr ISA intr-lst n)
             (< (j-i-1 addr ISA intr-lst n) n))
  :rule-classes :linear)

(defthm j-<-n
    (implies (ISA-self-modify-p ISA-0 intr-lst n)
```

```
            (< (j ISA-0 intr-lst n) n)))

; This lemma tells that
; (ISA-fetches-from addr ISA intr-lst n)
; => exists i j addr (and (< i j) (< j n)
;                         (ISA-writes-at addr ISA-i intr-i)
;                         (ISA-fetches-from addr ISA-j intr-j)
(defthm ISA-self-modify-p*
    (let ((i (i ISA-0 intr-lst n))
          (j (j ISA-0 intr-lst n))
          (addr (write-addr ISA-0 intr-lst n)))
      (let ((ISA-i (ISA-stepn ISA-0 intr-lst i))
            (intr-i (nth i intr-lst))
            (ISA-j (ISA-stepn ISA-0 intr-lst j))
            (intr-j (nth j intr-lst)))
        (implies (ISA-self-modify-p ISA-0 intr-lst n)
                 (and (ISA-writes-at addr ISA-i intr-i)
                      (ISA-fetches-from addr ISA-j intr-j)
                      (< i j) (< j n)))))
  :hints (("goal" :do-not-induct t)))

(defthm ISA-fetches-from-implies-ISA-stepn-fetches-from
    (implies (and (integerp j) (integerp n) (<= 0 j) (< j n)
                  (ISA-fetches-from addr (ISA-stepn ISA intr-lst j)
                                    (nth j intr-lst)))
             (ISA-stepn-fetches-from addr ISA intr-lst n))
  :hints (("goal" :in-theory (enable ISA-stepn))))

; This lemma tells that
; (ISA-fetches-from addr ISA intr-lst n)
; <= exists i j addr (and (< i j) (< j n)
;                         (ISA-writes-at addr ISA-i intr-i)
;                         (ISA-fetches-from addr ISA-j intr-j)
(defthm ISA-self-modify-p**
    (let ((ISA-i (ISA-stepn ISA-0 intr-lst i))
          (intr-i (nth i intr-lst))
          (ISA-j (ISA-stepn ISA-0 intr-lst j))
          (intr-j (nth j intr-lst)))
      (implies (and (integerp i) (integerp j) (integerp n) (<= 0 i)
                    (< i j) (< j n)
                    (ISA-writes-at addr ISA-i intr-i)
                    (ISA-fetches-from addr ISA-j intr-j))
               (ISA-self-modify-p ISA-0 intr-lst n)))
  :hints (("goal" :in-theory (enable ISA-WRITES-AT
                                     ISA-stepn
                                     ISA-STEPN-SELF-MODIFIES-P)
                  :restrict
                  ((ISA-fetches-from-implies-ISA-stepn-fetches-from
                    ((j (+ -1 j))))))))
|#

#|
Here is a simple example program for our ISA.

Our program calculates the factorial of the number at address #x800 and stores
it at address #x801.

Initial memory setting:

#x0: STI R0,(#x50)
#x1: LDI R0,(#x3)
#x2: BR R0, 0
```

```
#x3: 0

#x10: STI R0,(#x50)
#x11: LDI R0,(#x13)
#x12: BR R0, 0
#x13: 0

#x20: STI R0,(#x50)
#x21: LDI R0,(#x23)
#x22: BR R0, 0
#x23: 0

#x30: STI R0,(#x50)
#x31: LDI R0,(#x33)
#x32: BR R0, 0
#x33: 0

#x60: 0
#x61: 1
#x62: 2
#x63: -1

#x70: #x400
#x71: #x800

#x100: LDI R15,(#x70) ; program base
#x101: LDI R14,(#x71) ; data base
#x102: LDI R0, (#x60) ; 0
#x103: LDI R1, (#x61) ; 1
#x104: LDI R2, (#x62) ; 2
#x105: LDI R3, (#x63) ; -1
#x106: MTSR SR0,R15
#x107: MTSR SR1,R0
#x108: RFEH

Initial memory image:
#x400     LD R5,(R14+R0) ; R5 holds counter
#x401     ADD R6, R0, R1     ; R6 holds factorial. Initially 1.
Loop:
#x402:    MUL R6, R6, R5  ; counter * fact -> fact
#x403:    ADD R5, R5, R3  ; decrement fact
#x404:    BR  R5, Exit; if counter is zero, exit
#x405:    BR  R0, Loop ; always jump to loop
EXIT:
#x406: ST  R6, (R14+R1)
#x407: SYNC
#x408: Trap

#x800: 5
#x801: 0
#x802: 5 ; Offset to Loop
#x803: 9 ; Offset to Exit

(assign mem-alist '(
; Exception Handler
(#x0 . #x7050) ; STI R0,(#x50)
(#x1 . #x6003) ; LDI R0,(#x3)
(#x2 . #x2000) ; BR R0, 0
(#x3 . 0)
; Exception Handler
(#x10 . #x7050) ; STI R0,(#x50)
(#x11 . #x6013) ; LDI R0,(#x13)
```

```
(#x12 . #x2000) ; BR R0, 0
(#x13 . 0)
; Exception Handler
(#x20 . #x7050) ; STI R0,(#x50)
(#x21 . #x6023) ; LDI R0,(#x23)
(#x22 . #x2000) ; BR R0, 0
(#x23 . 0)

; Exception Handler
(#x30 . #x7050) ; STI R0,(#x50)
(#x31 . #x6033) ; LDI R0,(#x33)
(#x32 . #x2000) ; BR R0, 0
(#x33 . 0)

; Kernel Data Section
(#x60 .  0)
(#x61 . 1)
(#x62 . 2)
(#x63 . #xFFFF) ; -1
(#x70 . #x400)
(#x71 . #x800)
; Kernel Dispatching code
(#x100 . #x6F70) ; LDI R15,(#x70) ; program base
(#x101 . #x6E71) ;  LDI R14,(#x71) ; data base
(#x102 . #x6060) ;  LDI R0, (#x60) ; 0
(#x103 . #x6161) ;  LDI R1, (#x61) ; 1
(#x104 . #x6262) ; LDI R2, (#x62) ; 2
(#x105 . #x6363) ; LDI R3, (#x63) ; -1
(#x106 . #xAF00) ;  MTSR SR0,R15
(#x107 . #xA010) ;  MTSR SR1,R0
(#x108 . #x8000) ; #x103: RFEH
; Program
(#x400 . #x35E0) ;  LD R5,(R14+R0) ; R5 holds counter
(#x401 . #x0601) ;  ADD R6, R0, R1     ; R6 holds factorial. Initially 1.
; Loop:
(#x402 . #x1665) ;  Mul R6, R6, R5  ; counter * fact -> fact
(#x403 . #x0553) ;  ADD R5, R5, R3  ; decrement fact
(#x404 . #x2502) ;  BR  R5, Exit; if counter is zero, exit
(#x405 . #x20FD) ;  BR  R0, Loop ; always jump to loop
; EXIT:
(#x406 . #x46E1) ; ST  R6, (R14+R1)
(#x407 . #x5000) ; SYNC
(#x408 . #xB000) ; Trap

; Data Section
(#x800 . 5)
(#x801 . 0)
(#x802 . 5) ; Offset to Loop
(#x803 . 9) ; Offset to Exit
))

(assign mem (set-page-mode *read-only* 1 (compress1 'mem *init-mem*)))
(assign mem (set-page-mode *read-write* 2 (@ mem)))
(assign mem (compress1 'mem (load-mem-alist (@ mem-alist) (@ mem))))

How to run the program:
1. Certify and compile all the proof scripts.
   (You may skip this, but the execution will be slow.)
2. Run ACL2.
3. Type command '(ld "ISA-def.lisp")'.
4. Run following assign commands, which defines initial state s.
 (assign RF '(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
```

```
 (assign mem  See Above)
 (assign SRF (SRF 1 0 0))
 (assign s (ISA-state #x100 (@ RF) (@ SRF) (@ mem)))

5. You can run the ISA machine for one cycle by
   (ISA-step (@ s) (ISA-input 1)).
   You can also run the machine for multiple cycles with ISA-stepn.
   For instance, if you want to run the machine 15 cycles, type:
     (assign input-list (make-list 15 :initial-element (ISA-input 0)))
     (ISA-stepn (@ s) 15).

6. Following macro may be useful to evaluate and assign an ISA state
to a variable, and print out only pc and register file, but not memory.

(defmacro eval-set-print-ISA-RF (s expr)
  `(pprogn (f-put-global ',s ,expr state)
           (mv nil
             (list (ISA-pc (f-get-global ',s state))
                   (ISA-RF (f-get-global ',s state))
                   (ISA-SRF (f-get-global ',s state)))
              state)))

|#
```

D.2.3 MA2-def.tex

This file contains the definition of the FM9801 microarchitecture.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MA2-def.lisp
; Author  Jun Sawada, University of Texas at Austin
;
; This file contains the microarchitectural definition of the FM9801.
; This version deploys the Tomasulo Algorithm with a re-order buffer.
; This is an out-of-order multi-issue machine.  The specification is
; written using the IHS library.  It also requires the book basic-def,
; which defines the register file and the memory.
;
; The next-state function (MA-step MA sigs) returns the next state of
; the pipelined machine.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(in-package "ACL2")

(include-book "b-ops-aux-def")
(include-book "basic-def")
;(include-book "multiplier-def")

(deflabel begin-MA2-def)
(defconst *abstract-impl-flag* 'abstract)

(defun wrap-local (defs)
  (if (endp defs)
      nil
      (cons `(local ,(car defs)) (wrap-local (cdr defs)))))

(defmacro encapsulate-impl (name stubs defs thms)
  (let ((begin-def-label (pack-intern name "BEGIN-DEF-" name))
        (end-def-label (pack-intern name "END-DEF-" name))
        (def-label (pack-intern name "DEF-" name))
        (begin-theory-label (pack-intern name "BEGIN-THEORY-" name))
```

```
        (end-theory-label (pack-intern name "END-THEORY-" name))
        (theory-label (pack-intern name "THEORY-" name)))
    (cond ((equal *abstract-impl-flag* 'abstract)
           `(encapsulate ,stubs
            ,@(wrap-local defs)
            (deflabel ,begin-theory-label)
            ,@thms
            (deflabel ,end-theory-label)
            (deftheory ,theory-label
                (set-difference-theories
                 (universal-theory ',end-theory-label)
                 (universal-theory ',begin-theory-label)))))
          ((equal *abstract-impl-flag* 'executable)
           `(encapsulate nil
            (deflabel ,begin-def-label)
            ,@defs
            (deflabel ,end-def-label)
            (deflabel ,begin-theory-label)
            ,@thms
            (deflabel ,end-theory-label)
            (deftheory ,def-label
                (set-difference-theories
                 (universal-theory ',end-def-label)
                 (universal-theory ',begin-def-label)))
            (deftheory ,theory-label
                (set-difference-theories
                 (universal-theory ',end-theory-label)
                 (universal-theory ',begin-theory-label)))
            (in-theory (disable ,def-label))))
          (t
           (er hard 'encapsulate-impl
               "*abstract-impl-flag* must be 'abstract or 'executable")))))

(deflabel begin-MA-state)

(defconst *rob-index-size* 3)
(defconst *rob-size* (expt 2 *rob-index-size*))
(defbytetype rob-index *rob-index-size* :unsigned)

(defthm rob-index-p-forward-unsigned-byte
    (implies (rob-index-p idx) (unsigned-byte-p *rob-index-size* idx))
  :hints (("goal" :in-theory (enable rob-index-p)))
  :rule-classes :forward-chaining)

(defthm rob-index-p-bv-eqv-iff-equal
    (implies (and (rob-index-p idx1) (rob-index-p idx2))
             (equal (b1p (bv-eqv *rob-index-size* idx1 idx2))
                    (equal idx1 idx2)))
  :hints (("goal" :in-theory (enable rob-index-p))))

#|
THE DEFINITION OF CONTROL VECTOR:

The control bits are defined as follows:
 exunit: indicate the selected execution unit.
          exunit bit 0: integer unit
                 bit 1: multiplier
                 bit 2: load-store unit
                 bit 3: branch?
                 bit 4: no execution unit

 operand: indicates the operand format.
```

```
         bit 0: dispatcher outputs RA to port val1 and RB to port val2
         bit 1: dispatcher outputs immediate to port val1
         bit 2: dispatcher outputs RC to port val1
         bit 3: dispatcher outputs special register value to port val1
 br-predict?: branch is taken speculatively?
 ld-st?:  load or store when exunit=3.
                0: load
                1: store
 wb?:     Whether instruction needs to write back its result.
 wb-sreg?:    whether results should be written back to a special register.
 sync?:   whether the instruction forces synchronization.
 rfeh?:   This bit is set for RFEH instructions.

|#

;; The definition of the control vector. An instruction is decoded
;; into a control vector at the decode stage, and will be passed to
;; the following latches.
;;
;; Access functions to the fields are given by cntlv-<name of field>.
;; For instance, cntlv-exunit is the access function to field exunit.
(defword cntlv-word ((exunit 5 10)
                     (operand 4 6)
                     (br-predict? 1 5)
                     (ld-st? 1 4)
                     (wb? 1 3)
                     (wb-sreg? 1 2)
                     (sync? 1 1)
                     (rfeh? 1 0))
  :conc-name cntlv-)

(defbytetype cntlv 15 :unsigned)

#|
Exception Flags
 raised?:     Exception is raised
 ex-type: Exception type
                0: Illegal Instruction
                1: Fetch Error
                2: Data Access Error
                3: External Interrupt
|#
(defword excpt-word ((raised? 1 2)
                     (type 2 0))
  :conc-name excpt-)

(defbytetype excpt-flags 3 :unsigned)

;; Microarchitectural specification uses four inputs corresponding to
;; an external interrupt, a fetch unit response, a data unit response,
;; and the oracle to determine the branch prediction.  1 means the
;; corresponding event takes place.  For instance, an external
;; interrupt is requested when exintr is 1.  An instruction fetch can
;; take place when fetch is 1.  We try to model the asynchronous
;; behavior of memory with oracle inputs.  If we keep MA-input-fetch
;; to be 0, we can flush the pipeline.  Field br-predict is an oracle
;; to specify the nondeterministic behavior of the branch predictor.
(defstructure MA-input
  (exintr (:assert (bitp exintr) :rewrite))
  (br-predict (:assert (bitp br-predict) :rewrite))
  (fetch (:assert (bitp fetch) :rewrite))
  (data  (:assert (bitp data) :rewrite))
```

```
  (:options :guards))

(deflist MA-input-listp (l)
  (declare (xargs :guard t))
  MA-input-p)

; The instruction fetch unit.  The program counter is considered to
; belong to a different unit.  The instruction fetch unit stores an
; instruction temporarily, and sends it to the dispatching queue if
; there is an empty queue entry.
(defstructure IFU
  (valid? (:assert (bitp valid?) :rewrite))
  (excpt (:assert (excpt-flags-p excpt) :rewrite (:rewrite (integerp excpt))))
  (pc (:assert (addr-p pc) :rewrite (:rewrite (integerp pc))
               (:rewrite (acl2-numberp pc))))
  (word (:assert (word-p word) :rewrite (:rewrite (integerp word))))
  (:options :guards))

; A dispatch queue entry stores a decoded instruction.
(defstructure dispatch-entry
  (valid? (:assert (bitp valid?) :rewrite))
  (excpt (:assert (excpt-flags-p excpt) :rewrite (:rewrite (integerp excpt))))
  (pc (:assert (addr-p pc) :rewrite (:rewrite (integerp pc))
               (:rewrite (acl2-numberp pc))))
  (cntlv (:assert (cntlv-p cntlv) :rewrite (:rewrite (integerp cntlv))))
  (rc (:assert (rname-p rc) :rewrite))
  (ra (:assert (rname-p ra) :rewrite))
  (rb (:assert (rname-p rb) :rewrite))
  (im (:assert (immediate-p im) :rewrite))
  (br-target (:assert (addr-p br-target) :rewrite
                      (:rewrite (integerp br-target))))
  (:options :guards (:conc-name DE-)))

; Register Reference Table Entry.
; wait? is 1 iff the register value is not updated, and there are
; instructions in the ROB that modify this register.  Tag is the
; ROB entry that contains the instruction that modifies the corresponding
; register.
(defstructure reg-ref
  (wait? (:assert (bitp wait?) :rewrite))
  (tag (:assert (rob-index-p tag) :rewrite (:rewrite (integerp tag))))
  (:options :guards))

(deflist reg-ref-listp (l)
    (declare (xargs :guard t))
    reg-ref-p)

(defun reg-tbl-p (RF)
  (declare (xargs :guard t))
  (and (reg-ref-listp RF) (equal (len RF) *num-regs*)))

(defun reg-tbl-nth (n tbl)
  (declare (xargs :guard (and (rname-p n) (reg-tbl-p tbl))))
  (nth n tbl))

(defthm reg-ref-p-reg-tbl-nth
    (implies (and (reg-tbl-p tbl) (rname-p n))
             (reg-ref-p (reg-tbl-nth n tbl))))

(in-theory (disable reg-tbl-p reg-tbl-nth))

(defstructure sreg-tbl
```

```
  (sr0 (:assert (reg-ref-p sr0) :rewrite))
  (sr1 (:assert (reg-ref-p sr1) :rewrite))
  (:options :guards))

(defun sreg-tbl-nth (n tbl)
  (declare (xargs :guard (and (rname-p n) (sreg-tbl-p tbl))))
  (b-if (bv-eqv *rname-size* n 0) (sreg-tbl-sr0 tbl)
  (b-if (bv-eqv *rname-size* n 1) (sreg-tbl-sr1 tbl)
        (reg-ref 0 0))))

(defthm sreg-ref-p-sreg-tbl-nth
    (implies (and (sreg-tbl-p tbl) (rname-p n))
             (reg-ref-p (sreg-tbl-nth n tbl))))
(in-theory (disable sreg-tbl sreg-tbl-nth))

; Dispatch queue has 4 entries.
(defstructure DQ
  (DE0 (:assert (dispatch-entry-p DE0) :rewrite))
  (DE1 (:assert (dispatch-entry-p DE1) :rewrite))
  (DE2 (:assert (dispatch-entry-p DE2) :rewrite))
  (DE3 (:assert (dispatch-entry-p DE3) :rewrite))
  (reg-tbl (:assert (reg-tbl-p reg-tbl) :rewrite))
  (sreg-tbl (:assert (sreg-tbl-p sreg-tbl) :rewrite))
  (:options :guards))

(defthm len-DQ-reg-tbl
    (implies (DQ-p DQ)
             (equal (len (DQ-reg-tbl DQ)) *num-regs*))
  :hints (("Goal" :in-theory (enable reg-tbl-p DQ-p))))

; The ROB entry.
; excpt contains the exception flags.
; val stores the result of the corresponding instruction.
; dest is the destination register file.
; pc is the program counter of the instruction.
; br-predict? and br-actual? contains the branch prediction result and
;   actual outcome, respectively.
; Other fields are control vector fields.
(defstructure ROB-entry
  (valid? (:assert (bitp valid?) :rewrite))
  (complete? (:assert (bitp complete?) :rewrite))
  (excpt (:assert (excpt-flags-p excpt) :rewrite (:rewrite (integerp excpt))))
  (wb? (:assert (bitp wb?) :rewrite))
  (wb-sreg? (:assert (bitp wb-sreg?) :rewrite))
  (sync? (:assert (bitp sync?) :rewrite))
  (branch? (:assert (bitp branch?) :rewrite))
  (rfeh? (:assert (bitp rfeh?) :rewrite))
  (br-predict? (:assert (bitp br-predict?) :rewrite))
  (br-actual? (:assert (bitp br-actual?) :rewrite))
  (pc (:assert (addr-p pc) :rewrite (:rewrite (integerp pc))
               (:rewrite (acl2-numberp pc))))
  (val (:assert (word-p val) :rewrite (:rewrite (integerp val))))
  (dest (:assert (rname-p dest) :rewrite (:rewrite (integerp dest))))
  (:options :guards (:conc-name ROBE-)))

(deflist ROBE-listp (l)
  (declare (xargs :guard t))
  ROB-entry-p)

(defun ROB-entries-p (l)
  (declare (xargs :guard t))
  (and (ROBE-listp l) (equal (len l) *rob-size*)))
```

```
(defthm true-listp-rob-entries
    (implies (ROB-entries-p l) (true-listp l)))

(in-theory (disable ROB-entries-p))

; Reorder buffer contains re-order buffer entries and indices to their
; head and tail. Re-order buffer is a FIFO queue.  Head points to the
; oldest instruction which will be committed next.  Tail points to the
; newest instruction which is dispatched most recently.  It also
; contains the external interrupt flag exintr? and a flag to indicate
; whether head is larger than equal to tail.  This flag is set when
; the tail is wrapped-around to 0, and it is cleared when the head is
; wrapped-around to 0.  If head and tail are equal and flag is off,
; the reorder buffer is empty.  If head and tail are equal and flag is
; on, the reorder buffer is full.
;
;
;        -------                  -------
;      0|       | flg=0       0|  I2   |  flg=1
;      1|       |              1|  I3   |
;      2|-------|              2|-------|
;      3|   I1  |<- HEAD       3|       |<- TAIL
;      4|   I2  |              4|       |
;      5|   I3  |              5|       |
;      6|-------|              6|-------|
;      7|       |<- TAIL       7|  I1   |<- HEAD
;        -------                  -------
(defstructure ROB
  (flg (:assert (bitp flg) :rewrite))
  (exintr? (:assert (bitp exintr?) :rewrite))
  (head (:assert (rob-index-p head) :rewrite
                 (:type-prescription (integerp head))))
  (tail (:assert (rob-index-p tail) :rewrite
                 (:type-prescription (integerp tail))))
  (entries (:assert (rob-entries-p entries) :rewrite))
  (:options :guards))

(defthm rob-listp-rob-entries
    (implies (rob-p rob) (robe-listp (rob-entries rob)))
  :hints (("Goal" :in-theory (enable rob-entries-p)
                  :use (defs-rob-assertions))))

(defun nth-ROBE (n ROB)
  (declare (xargs :guard (and (rob-index-p n) (ROB-p ROB))))
  (nth n (ROB-entries ROB)))

(defthm ROB-entry-p-nth-rob-entries
    (implies (and (rob-index-p n) (ROB-entries-p rob-entries))
             (ROB-entry-p (nth n rob-entries)))
  :hints (("goal" :in-theory (enable rob-index-p rob-entries-p))))

(defthm ROB-entry-p-nth-robe
    (implies (and (rob-index-p n) (ROB-p ROB))
             (ROB-entry-p (nth-ROBE n ROB))))

(in-theory (disable nth-ROBE))

; The structure of a reservation station.
; Op indicates what type of integer operation should be performed.
; When Op=0, IU returns the sum of two operands. When Op=1, IU returns the
; the first value itself.
```

```
(defstructure RS
  (valid? (:assert (bitp valid?) :rewrite))
  (op (:assert (bitp op) :rewrite))
  (tag (:assert (rob-index-p tag) :rewrite))
  (ready1? (:assert (bitp ready1?) :rewrite))
  (ready2? (:assert (bitp ready2?) :rewrite))
  (val1 (:assert (word-p val1) :rewrite (:rewrite (integerp val1))
                 (:rewrite (acl2-numberp val1))))
  (val2 (:assert (word-p val2) :rewrite (:rewrite (integerp val2))
                 (:rewrite (acl2-numberp val2))))
  (src1 (:assert (rob-index-p src1) :rewrite))
  (src2 (:assert (rob-index-p src2) :rewrite))
  (:options :guards))

; The IU unit. It has two reservation stations.
(defstructure integer-unit
  (rs0 (:assert (RS-p rs0) :rewrite))
  (rs1 (:assert (RS-p rs1) :rewrite))
  (:options :guards (:conc-name IU-)))

; Latch of the MU.  tag is the Tomasulo's tag for the instruction in
; the latch. data contains the intermediate value.
;
; Note:
; The following definition of the multiplier latch is used for the
; when plugging in the real multiplier in multiplier-def.lisp.
;(defstructure MU-latch1
;  (valid? (:assert (bitp valid?) :rewrite))
;  (tag (:assert (rob-index-p tag) :rewrite))
;  (data (:assert (ML1-data-p data) :rewrite))
;  (:options :guards))

(defstructure MU-latch1
  (valid? (:assert (bitp valid?) :rewrite))
  (tag (:assert (rob-index-p tag) :rewrite))
  (data)
  (:options :guards))

; Note:
; The following definition of the multiplier latch is used for the
; when plugging in the real multiplier.
;(defstructure MU-latch2
;  (valid? (:assert (bitp valid?) :rewrite))
;  (tag (:assert (rob-index-p tag) :rewrite))
;  (data (:assert (ML2-data-p data) :rewrite))
;  (:options :guards))

(defstructure MU-latch2
  (valid? (:assert (bitp valid?) :rewrite))
  (tag (:assert (rob-index-p tag) :rewrite))
  (data)
  (:options :guards))

; Three stage multiplier unit.
(defstructure mult-unit
  (rs0 (:assert (RS-p rs0) :rewrite))
  (rs1 (:assert (RS-p rs1) :rewrite))
  (lch1 (:assert (MU-latch1-p lch1) :rewrite))
  (lch2 (:assert (MU-latch2-p lch2) :rewrite))
  (:options :guards (:conc-name MU-)))

; The structure of a reservation station for the memory unit.
```

```
; Op indicates how to calculate the access address. 0 means the sum of
; RA and RB registers is the access address.  1 means that the immediate
; value is the access address.
(defstructure LSU-RS
  (valid? (:assert (bitp valid?) :rewrite))
  (op (:assert (bitp op) :rewrite))
  (ld-st? (:assert (bitp ld-st?) :rewrite))
  (tag (:assert (rob-index-p tag) :rewrite))
  (rdy3? (:assert (bitp rdy3?) :rewrite))
  (val3 (:assert (word-p val3) :rewrite (:rewrite (integerp val3))))
  (src3 (:assert (rob-index-p src3) :rewrite))
  (rdy1? (:assert (bitp rdy1?) :rewrite))
  (val1 (:assert (word-p val1) :rewrite (:rewrite (integerp val1))
                (:rewrite (acl2-numberp val1))))
  (src1 (:assert (rob-index-p src1) :rewrite))
  (rdy2? (:assert (bitp rdy2?) :rewrite))
  (val2 (:assert (word-p val2) :rewrite (:rewrite (integerp val2))
                (:rewrite (acl2-numberp val2))))
  (src2 (:assert (rob-index-p src2) :rewrite))
  (:options :guards))

; Fields of the read buffer.  Fields wbuf0? and wbuf1? records whether
; write buffer 0 and 1 are occupied when the read operation is issued.
; In other words, wbuf0? implies that the instruction at wbuf0
; precedes the read operation in program order.  This is used to check
; whether data dependencies exist between load and store instructions.
; tag is Tomasulo's tag.  Addr is the access address.
(defstructure read-buffer
  (valid? (:assert (bitp valid?) :rewrite))
  (tag (:assert (rob-index-p tag) :rewrite))
  (addr (:assert (addr-p addr) :rewrite (:rewrite (integerp addr))))
  (wbuf0? (:assert (bitp wbuf0?) :rewrite))
  (wbuf1? (:assert (bitp wbuf1?) :rewrite))
  (:options :guards (:conc-name rbuf-)))

; Write buffer entry.  Valid? is 1, whenever it contain a valid
; instruction.  complete? is 1 when the address check is performed.
; commit? is set to 1, when the corresponding instruction commits.
; The write operation is not performed until the commit occurs,
; because there is no way to roll back the memory access when an
; exception occurs.  Tag is Tomasulo's tag.  Addr is the access
; address.  Val is the stored instruction.
(defstructure write-buffer
  (valid? (:assert (bitp valid?) :rewrite))
  (complete? (:assert (bitp complete?) :rewrite))
  (commit? (:assert (bitp commit?) :rewrite))
  (tag (:assert (rob-index-p tag) :rewrite))
  (addr (:assert (addr-p addr) :rewrite (:rewrite (integerp addr))))
  (val (:assert (word-p val) :rewrite (:rewrite (integerp val))))
  (:options :guards (:conc-name wbuf-)))

; The result latch for the LSU.  excpt contains the exception flag.
; tag is Tomasulo's tag.  val is the load instruction result.
(defstructure LSU-latch
  (valid? (:assert (bitp valid?) :rewrite))
  (excpt (:assert (excpt-flags-p excpt) :rewrite (:rewrite (integerp excpt))))
  (tag (:assert (rob-index-p tag) :rewrite))
  (val (:assert (word-p val) :rewrite (:rewrite (integerp val))))
  (:options :guards))

; The reservation stations for the LSU forms a queue, because the
; memory access order is critical in executing programs.  The head of the
```

```
; queue is indicated by the flag rs1-head?.  When rs1-head? is on, RS1 is
; the head of the queue.  Otherwise, RS0 is the head.
(defstructure load-store-unit
  (rs1-head? (:assert (bitp rs1-head?) :rewrite))
  (rs0 (:assert (LSU-RS-p rs0) :rewrite))
  (rs1 (:assert (LSU-RS-p rs1) :rewrite))
  (rbuf (:assert (read-buffer-p rbuf) :rewrite))
  (wbuf0 (:assert (write-buffer-p wbuf0) :rewrite))
  (wbuf1 (:assert (write-buffer-p wbuf1) :rewrite))
  (lch (:assert (LSU-latch-p lch) :rewrite))
  (:options :guards (:conc-name LSU-)))

; The reservation station for the BU stores the operand RC.  BR
; command checks whether RC is equal to 0, and if so it sets the
; program counter to the sum of the program counter for the BR
; instruction and the relative address obtained from the im field.
; This jump address is calculated in the decode stage.
(defstructure BU-RS
  (valid? (:assert (bitp valid?) :rewrite))
  (tag (:assert (rob-index-p tag) :rewrite))
  (ready? (:assert (bitp ready?) :rewrite))
  (val (:assert (word-p val) :rewrite (:rewrite (integerp val))))
  (src (:assert (rob-index-p src) :rewrite))
  (:options :guards))

; The BU has two reservation stations.
(defstructure branch-unit
  (rs0 (:assert (BU-RS-p rs0) :rewrite))
  (rs1 (:assert (BU-RS-p rs1) :rewrite))
  (:options :guards (:conc-name BU-)))

;; Definition of the pipelined machine states. A machine state
;; contains a program counter, a register file, special register file,
;; instruction fetch unit, dispatch unit, re-order buffer, integer
;; unit, multiplier unit branch unit memory unit and memory.
(defstructure MA-state
  (pc (:assert (addr-p pc) :rewrite (:rewrite (integerp pc))
               (:rewrite (acl2-numberp pc))))
  (RF (:assert (RF-p RF) :rewrite))
  (SRF (:assert (SRF-p SRF) :rewrite))
  (IFU  (:assert (IFU-p IFU) :rewrite))
  (DQ   (:assert (DQ-p DQ) :rewrite))
  (ROB (:assert (rob-p ROB) :rewrite :type-prescription))
  (IU  (:assert (integer-unit-p IU) :rewrite))
  (MU  (:assert (mult-unit-p MU) :rewrite))
  (BU (:assert (branch-unit-p BU) :rewrite))
  (LSU (:assert (load-store-unit-p LSU) :rewrite))
  (mem (:assert (mem-p mem) :rewrite))
  (:options :guards (:conc-name MA-)))

(deflabel end-MA-state)

(deflabel begin-MA-def)

; This is the branch prediction function.  We can replace the
; implementation with other realistic ones.
(encapsulate-impl branch-predict
   ((branch-predict? (IFU MA sigs) t))
((defun branch-predict? (IFU MA sigs)
   (declare (xargs :guard (and (IFU-p IFU) (MA-state-p MA) (MA-input-p sigs))))
   (MA-input-br-predict sigs)))
```

```
((defthm bitp-branch-predict?
     (implies (MA-input-p sigs) (bitp (branch-predict? IFU MA sigs))))))

; IFU-branch-predict? is set if the decode unit finds a branch instruction
; in IFU and it predicts the branch is taken.  IFU-branch-target should
; post the target address when IFU-branch-predict? is set.
(defun IFU-branch-predict? (IFU MA sigs)
  (declare (xargs :guard (and (IFU-p IFU) (MA-state-p MA) (MA-input-p sigs))))
  (bs-and (IFU-valid? IFU)
          (b-not (excpt-raised? (IFU-excpt IFU)))
          (bv-eqv *opcode-size* (opcode (IFU-word IFU)) 2)
          (branch-predict? IFU MA sigs)))

(defthm bitp-IFU-branch-predict (bitp (IFU-branch-predict? IFU MA sigs)))

; Generating the branch target address.
(defun IFU-branch-target (IFU)
  (declare (xargs :guard (IFU-p IFU)))
  (addr (+ (IFU-pc IFU)
           (logextu *addr-size* *immediate-size* (im (IFU-word IFU))))))

(defthm addr-p-IFU-branch-target
    (implies (IFU-p IFU) (addr-p (IFU-branch-target IFU))))
(in-theory (disable IFU-branch-target))

; Instruction fetch is prohibited, if the memory is not readable, and
; the processor is in the user mode.
(defun IFU-fetch-prohibited? (pc mem su)
  (declare (xargs :guard (and (addr-p pc) (mem-p mem) (bitp su))))
  (b-nor (readable-addr? pc mem) su))

#|

THE DEFINITION OF CONTROL VECTOR:

The control bits are defined as follows:
 exunit: indicate the selected execution unit.
          exunit bit 0: adder
                 bit 1: multiplier
                 bit 2: load-store unit
                 bit 3: branch?
                 bit 4: no execution unit
 operand: indicates the operand format.
          bit 0: dispatcher outputs RA to port val1 and RB to port val2
          bit 1: dispatcher outputs immediate to port val1
          bit 2: dispatcher outputs RC to port val1
          bit 3: dispatcher outputs special register value to port val1
 branch-predict?: branch is taken speculatively?
 ld-st?:  load or store when exunit=3.
                 0: load
                 1: store
 wb?:     Whether instruction needs to write back its result.
 wb-sreg?:    whether results should be written back to a special register.
 sync?:   whether the instruction forces a synchronization.
 rfeh?:   This bit is on for instruction RFEH

We define six opcodes:

 ADD 0          Addition
 MUL 1          Multiplication
 BR  2          Conditional Branch
 LD  3          Load Memory
```

```
 ST  4          Store Memory
 SYNC 5         Synchronize
 LD-IM 6        Load from an immediate address
 ST-IM 7        Store at an immediate address
 RFEH  8        Return from Exception Handling (privileged)
 MFSR  9        Move From a Special Register (privileged)
 MTSR  10       Move To a Special Register (privileged)
|#

; Decoder.
(defun decode (opcd branch-predict?)
  (declare (xargs :guard (and (opcd-p opcd) (bitp branch-predict?))
                  :guard-hints (("Goal" :in-theory (enable opcd-p)))))
  (logcons (bv-eqv *opcode-size* opcd 8) ; rfeh?
  (logcons (b-ior (bv-eqv *opcode-size* opcd 5)
                  (bv-eqv *opcode-size* opcd 8)) ; sync?
  (logcons (bv-eqv *opcode-size* opcd 10) ; wb-sreg?
  (logcons (bs-ior (bv-eqv *opcode-size* opcd 0)
                   (bv-eqv *opcode-size* opcd 1)
                   (bv-eqv *opcode-size* opcd 3)
                   (bv-eqv *opcode-size* opcd 6)
                   (bv-eqv *opcode-size* opcd 9)
                   (bv-eqv *opcode-size* opcd 10)) ; wb?
  (logcons (b-ior (bv-eqv *opcode-size* opcd 4)
                  (bv-eqv *opcode-size* opcd 7)) ; ld-st?
  (logcons branch-predict? ; branch-predict?
  (logcons (bs-ior (bv-eqv *opcode-size* opcd 0)
                   (bv-eqv *opcode-size* opcd 1)
                   (bv-eqv *opcode-size* opcd 3)
                   (bv-eqv *opcode-size* opcd 4)) ; operand:0
  (logcons (b-ior (bv-eqv *opcode-size* opcd 6)
                  (bv-eqv *opcode-size* opcd 7)) ; operand:1
  (logcons (b-ior (bv-eqv *opcode-size* opcd 2)
                  (bv-eqv *opcode-size* opcd 10)) ; operand:2
  (logcons (bv-eqv *opcode-size* opcd 9) ;operand:3
  (logcons (bs-ior (bv-eqv *opcode-size* opcd 0)
                   (bv-eqv *opcode-size* opcd 9)
                   (bv-eqv *opcode-size* opcd 10)) ; exunit:0
  (logcons (bv-eqv *opcode-size* opcd 1) ; exunit:1
  (logcons (bs-ior (bv-eqv *opcode-size* opcd 3)
                   (bv-eqv *opcode-size* opcd 4)
                   (bv-eqv *opcode-size* opcd 6)
                   (bv-eqv *opcode-size* opcd 7))
  (logcons (bv-eqv *opcode-size* opcd 2)
           (b-ior (bv-eqv *opcode-size* opcd 5)
                  (bv-eqv *opcode-size* opcd 8)))))))))))))))) ; exunit:2

(defthm cntlv-p-decode
    (implies (and (opcd-p opcd) (bitp flg)) (cntlv-p (decode opcd flg)))
  :hints (("Goal" :in-theory (enable cntlv-p unsigned-byte-p*))))

(in-theory (disable decode))

; Whether the decoded instruction is an illegal instruction.
(defun decode-illegal-inst? (opcd su ra)
  (declare (xargs :guard (and (opcd-p opcd) (bitp su) (rname-p ra))
                  :guard-hints (("Goal" :in-theory (enable opcd-p)))))
  (bs-and (b-not (bv-eqv *opcode-size* opcd 0))
          (b-not (bv-eqv *opcode-size* opcd 1))
          (b-not (bv-eqv *opcode-size* opcd 2))
          (b-not (bv-eqv *opcode-size* opcd 3))
          (b-not (bv-eqv *opcode-size* opcd 4))
```

```
          (b-not (bv-eqv *opcode-size* opcd 5))
          (b-not (bv-eqv *opcode-size* opcd 6))
          (b-not (bv-eqv *opcode-size* opcd 7))
          (b-not (b-and (bv-eqv *opcode-size* opcd 8) su))
          (b-not (bs-and (bv-eqv *opcode-size* opcd 9)
                         su
                         (b-ior (bv-eqv *rname-size* ra 0)
                                (bv-eqv *rname-size* ra 1))))
          (b-not (bs-and (bv-eqv *opcode-size* opcd 10)
                         su
                         (b-ior (bv-eqv *rname-size* ra 0)
                                (bv-eqv *rname-size* ra 1))))))

(defthm bitp-decode-illegal-inst (bitp (decode-illegal-inst? opcd su ra)))

; Output from the decoder.
(defun decode-output (IFU MA sigs)
  (declare (xargs :guard (and (IFU-p IFU) (MA-state-p MA) (MA-input-p sigs))
                  :verify-guards nil))
  (dispatch-entry (IFU-valid? IFU)
                  (b-if (excpt-raised? (IFU-excpt IFU))
                        (IFU-excpt IFU)
                        (b-if (decode-illegal-inst?
                               (opcode (IFU-word IFU)) (SRF-su (MA-SRF MA))
                               (ra (IFU-word IFU)))
                              #b100 0))
                  (IFU-pc IFU)
                  (decode (opcode (IFU-word IFU))
                          (IFU-branch-predict? IFU MA sigs))
                  (rc (IFU-word IFU))
                  (ra (IFU-word IFU))
                  (rb (IFU-word IFU))
                  (im (IFU-word IFU))
                  (IFU-branch-target IFU)))

(verify-guards decode-output
    :hints (("Goal" :in-theory (enable opcd-p))))

(defthm dispatch-entry-p-decode-output
    (implies (and (IFU-p IFU) (MA-state-p MA) (MA-input-p sigs))
             (dispatch-entry-p (decode-output IFU MA sigs))))

(in-theory (disable decode-output))

; The destination register of the dispatched instruction.
(defun DQ-out-dest-reg (DQ)
  (declare (xargs :guard (DQ-p DQ)))
  (let ((DE0 (DQ-DE0 DQ)))
    (b-if (cntlv-wb-sreg? (DE-cntlv DE0))
          (DE-ra DE0)
          (DE-rc DE0))))

; Whether the first operand is ready.
(defun DQ-out-ready1? (DQ)
  (declare (xargs :guard (DQ-p DQ)))
  (let ((DE0 (DQ-DE0 DQ))
        (cntlv (DE-cntlv (DQ-DE0 DQ))))
    (b-if (logbit 0 (cntlv-operand cntlv))
          (b-not (reg-ref-wait? (reg-tbl-nth (DE-ra DE0) (DQ-reg-tbl DQ))))
    (b-if (logbit 1 (cntlv-operand cntlv)) 1
    (b-if (logbit 2 (cntlv-operand cntlv))
          (b-not (reg-ref-wait? (reg-tbl-nth (DE-rc DE0) (DQ-reg-tbl DQ))))
```

```
    (b-if (logbit 3 (cntlv-operand cntlv))
          (b-not (reg-ref-wait?
                  (sreg-tbl-nth (DE-ra DE0) (DQ-sreg-tbl DQ))))
          0))))))

(defthm bitp-DQ-out-ready1
    (implies (DQ-p DQ) (bitp (DQ-out-ready1? DQ))))

(in-theory (disable DQ-out-ready1?))

; Tomasulo's tag of the instruction that produces the first operand.
(defun DQ-out-tag1 (DQ)
  (declare (xargs :guard (DQ-p DQ)))
  (let ((DE0 (DQ-DE0 DQ)) (cntlv (DE-cntlv (DQ-DE0 DQ))))
    (b-if (logbit 0 (cntlv-operand cntlv))
          (reg-ref-tag (reg-tbl-nth (DE-ra DE0) (DQ-reg-tbl DQ)))
    (b-if (logbit 2 (cntlv-operand cntlv))
          (reg-ref-tag (reg-tbl-nth (DE-rc DE0) (DQ-reg-tbl DQ)))
    (b-if (logbit 3 (cntlv-operand cntlv))
          (reg-ref-tag (sreg-tbl-nth (DE-ra DE0) (DQ-sreg-tbl DQ)))
          0)))))

(defthm rob-index-p-DQ-out-tag1
    (implies (DQ-p DQ) (rob-index-p (DQ-out-tag1 DQ)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (DQ-p DQ) (integerp (DQ-out-tag1 DQ))))))

(in-theory (disable DQ-out-tag1))

; DQ-read-val1 reads the value of the first operand from the corresponding
; dispatch queue entry or register file.
(defun DQ-read-val1 (DQ MA)
  (declare (xargs :guard (and (DQ-p DQ) (MA-state-p MA))))
  (let ((RF (MA-RF MA))
        (SRF (MA-SRF MA))
        (DE0 (DQ-DE0 DQ))
        (cntlv (DE-cntlv (DQ-DE0 DQ))))
    (b-if (logbit 0 (cntlv-operand cntlv)) (read-reg (DE-ra DE0) RF)
    (b-if (logbit 1 (cntlv-operand cntlv)) (word (DE-im DE0))
    (b-if (logbit 2 (cntlv-operand cntlv)) (read-reg (DE-rc DE0) RF)
    (b-if (logbit 3 (cntlv-operand cntlv)) (read-sreg (DE-ra DE0) SRF)
          0))))))

(defthm word-p-DQ-read-val1
    (implies (and (DQ-p DQ) (MA-state-p MA))
             (word-p (DQ-read-val1 DQ MA))))
(in-theory (disable DQ-read-val1))

; Whether the second operand value is ready.
(defun DQ-out-ready2? (DQ)
  (declare (xargs :guard (DQ-p DQ)))
  (b-not (reg-ref-wait? (reg-tbl-nth (DE-rb (DQ-DE0 DQ))
                                     (DQ-reg-tbl DQ)))))

(defthm bitp-DQ-out-ready2
    (implies (DQ-p DQ) (bitp (DQ-out-ready2? DQ))))

(in-theory (disable DQ-out-ready2?))

; The second operand register designator.
```

```
(defun DQ-out-reg2 (DQ)
  (declare (xargs :guard (DQ-p DQ)))
  (DE-rb (DQ-DE0 DQ)))

(defthm rname-p-DQ-out-reg2
    (implies (DQ-p DQ) (rname-p (DQ-out-reg2 DQ))))

(in-theory (disable DQ-out-reg2))

; Tomasulo's tag for the instruction that produces the second operand.
(defun DQ-out-tag2 (DQ)
  (declare (xargs :guard (DQ-p DQ)))
  (reg-ref-tag (reg-tbl-nth (DE-rb (DQ-DE0 DQ)) (DQ-reg-tbl DQ))))

(defthm rob-index-p-DQ-out-tag2
    (implies (DQ-p DQ) (rob-index-p (DQ-out-tag2 DQ)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (DQ-p DQ) (integerp (DQ-out-tag2 DQ))))))

(in-theory (disable DQ-out-tag2))

; Whether the third operand is ready.
(defun DQ-out-ready3? (DQ)
  (declare (xargs :guard (DQ-p DQ)))
  (b-not (reg-ref-wait? (reg-tbl-nth (DE-rc (DQ-DE0 DQ))
                                     (DQ-reg-tbl DQ)))))

(defthm bitp-DQ-out-ready3
    (implies (DQ-p DQ) (bitp (DQ-out-ready3? DQ))))

(in-theory (disable DQ-out-ready3?))

; The third operand register designator.
(defun DQ-out-reg3 (DQ)
  (declare (xargs :guard (DQ-p DQ)))
  (DE-rc (DQ-DE0 DQ)))

(defthm rname-p-DQ-out-reg3
    (implies (DQ-p DQ) (rname-p (DQ-out-reg3 DQ))))

(in-theory (disable DQ-out-reg3))

; Tomasulo's tag for the instruction that produces the third operand value.
(defun DQ-out-tag3 (DQ)
  (declare (xargs :guard (DQ-p DQ)))
  (reg-ref-tag (reg-tbl-nth (DE-rc (DQ-DE0 DQ)) (DQ-reg-tbl DQ))))

(defthm rob-index-p-DQ-out-tag3
    (implies (DQ-p DQ) (rob-index-p (DQ-out-tag3 DQ)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (DQ-p DQ) (integerp (DQ-out-tag3 DQ))))))

(in-theory (disable DQ-out-tag3))

; DQ-full? is set when dispatch queue is full.  The fetching is stalled
; until DQ has an available slot.
(defun DQ-full? (DQ)
  (declare (xargs :guard (and (DQ-p DQ))))
```

```
  (DE-valid? (DQ-DE3 DQ)))

(defthm bitp-DQ-full? (implies (DQ-p DQ) (bitp (DQ-full? DQ))))
(in-theory (disable DQ-full?))

; Ready to dispatch an instruction that goes directly into the complete
; stage. E.g., an illegal instruction.
(defun DQ-ready-no-unit? (DQ)
  (declare (xargs :guard (DQ-p DQ)))
  (b-and (DE-valid? (DQ-DE0 DQ))
         (b-ior (logbit 4 (cntlv-exunit (DE-cntlv (DQ-DE0 DQ))))
                (excpt-raised? (DE-excpt (DQ-DE0 DQ))))))

(defthm bitp-DQ-ready-no-unit (bitp (DQ-ready-no-unit? DQ)))

; Ready to dispatch to IU.
(defun DQ-ready-to-IU? (DQ)
  (declare (xargs :guard (DQ-p DQ)))
  (bs-and (DE-valid? (DQ-DE0 DQ))
          (logbit 0 (cntlv-exunit (DE-cntlv (DQ-DE0 DQ))))
          (b-not (excpt-raised? (DE-excpt (DQ-DE0 DQ))))))

(defthm bitp-DQ-ready-to-IU (bitp (DQ-ready-to-IU? DQ)))

; Ready to dispatch to MU.
(defun DQ-ready-to-MU? (DQ)
  (declare (xargs :guard (DQ-p DQ)))
  (bs-and (DE-valid? (DQ-DE0 DQ))
          (logbit 1 (cntlv-exunit (DE-cntlv (DQ-DE0 DQ))))
          (b-not (excpt-raised? (DE-excpt (DQ-DE0 DQ))))))

(defthm bitp-DQ-ready-to-MU (bitp (DQ-ready-to-MU? DQ)))

; Ready to dispatch to LSU.
(defun DQ-ready-to-LSU? (DQ)
  (declare (xargs :guard (DQ-p DQ)))
  (bs-and (DE-valid? (DQ-DE0 DQ))
          (logbit 2 (cntlv-exunit (DE-cntlv (DQ-DE0 DQ))))
          (b-not (excpt-raised? (DE-excpt (DQ-DE0 DQ))))))

(defthm bitp-DQ-ready-to-LSU (bitp (DQ-ready-to-LSU? DQ)))

; Ready to dispatch to BU.
(defun DQ-ready-to-BU? (DQ)
  (declare (xargs :guard (DQ-p DQ)))
  (bs-and (DE-valid? (DQ-DE0 DQ))
          (logbit 3 (cntlv-exunit (DE-cntlv (DQ-DE0 DQ))))
          (b-not (excpt-raised? (DE-excpt (DQ-DE0 DQ))))))

(defthm bitp-DQ-ready-to-BU (bitp (DQ-ready-to-BU? DQ)))

; If the ROB-flg is set, and ROB-head and ROB-tail are equal to each other,
; then the reorder buffer is full.
(defun ROB-full? (ROB)
  (declare (xargs :guard (ROB-p ROB)))
  (b-and (ROB-flg ROB)
         (bv-eqv *rob-index-size* (ROB-head ROB) (ROB-tail ROB))))

(defthm bitp-ROB-full (bitp (ROB-full? ROB)))

; If the ROB-flg is unset, and ROB-head and ROB-tail are equal to each other,
; then the reorder buffer is empty.
```

```
(defun ROB-empty? (ROB)
  (declare (xargs :guard (ROB-p ROB)))
  (bs-and (b-not (ROB-flg ROB))
          (bv-eqv *rob-index-size* (ROB-head ROB) (ROB-tail ROB))))

(defthm bitp-ROB-empty (bitp (ROB-empty? ROB)))

; The ROB entry designated by idx is empty.
(defun robe-empty? (idx rob)
  (declare (xargs :guard (and (ROB-p ROB) (rob-index-p idx))))
  (b-not (robe-valid? (nth-robe idx rob))))

(defun robe-empty-under? (idx ROB)
  (declare (xargs :guard (and (ROB-p ROB) (integerp idx)
                              (<= 0 idx) (<= idx *rob-size*))
                  :verify-guards nil))
  (if (zp idx)
      1
      (b-and (robe-empty? (1- idx) ROB)
             (robe-empty-under? (1- idx) ROB))))

(defthm bitp-robe-empty-under
    (bitp (robe-empty-under? idx ROB)))

(verify-guards robe-empty-under?
               :hints (("goal" :In-theory (enable rob-index-p
                                                  unsigned-byte-p))))

; The ROB is empty.
(defun ROB-entries-empty? (ROB)
  (declare (xargs :guard (ROB-p ROB)))
  (robe-empty-under? *rob-size* ROB))

; There are pending write operation in the write buffer.
(defun LSU-pending-writes? (LSU)
  (declare (xargs :guard (load-store-unit-p LSU)))
  (let ((wbuf0 (LSU-wbuf0 LSU)) (wbuf1 (LSU-wbuf1 LSU)))
    (b-ior (b-and (wbuf-valid? wbuf0) (wbuf-commit? wbuf0))
           (b-and (wbuf-valid? wbuf1) (wbuf-commit? wbuf1)))))

(defthm bitp-LSU-pending-writes (bitp (LSU-pending-writes? LSU)))

; This line coming out of ROB is raised if the ROB detects a branch
; misprediction and synchronization.  When entering an exception or leaving
; one, this line may not be raised.
(defun commit-jmp? (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (let ((LSU (MA-LSU MA)) (ROB (MA-ROB MA)))
    (let ((ROBE (nth-ROBE (ROB-head ROB) ROB)))
      (bs-and (ROBE-valid? ROBE)
              (ROBE-complete? ROBE)
              (b-ior (b-andc2 (ROBE-sync? ROBE)
                              (LSU-pending-writes? LSU))
                     (bs-and (ROBE-branch? ROBE)
; This line is added to be conservative.  Without it
; we don't know whether the implementation is correct.
                             (b-not (excpt-raised? (ROBE-excpt robe)))
                             (b-xor (ROBE-br-predict? ROBE)
                                    (ROBE-br-actual? ROBE))))))))

(defthm bitp-commit-jmp (bitp (commit-jmp? MA)))
(in-theory (disable commit-jmp?))
```

```
; Enter-excpt? is raised if an internal exception is raised.
(defun enter-excpt? (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (let ((robe (nth-ROBE (ROB-head (MA-ROB MA)) (MA-ROB MA)))
        (LSU (MA-LSU MA)))
    (bs-and (ROBE-valid? robe)
            (ROBE-complete? robe)
            (excpt-raised? (ROBE-excpt robe))
            (b-not (LSU-pending-writes? LSU)))))

(defthm bitp-enter-excpt (bitp (enter-excpt? MA)))
(in-theory (disable enter-excpt?))

; Ex-intr? is raised if the external exception handling starts.
(defun ex-intr? (MA sigs)
  (declare (xargs :guard (and (MA-state-p MA) (MA-input-p sigs))))
  (let ((ROB (MA-ROB MA)) (LSU (MA-LSU MA)))
    (bs-and (ROB-empty? ROB)
            (b-not (LSU-pending-writes? LSU))
            (b-ior (ROB-exintr? ROB)
                   (MA-input-exintr sigs)))))

(defthm bitp-ex-intr (bitp (ex-intr? MA sigs)))
(in-theory (disable ex-intr?))

; External interrupt address.
(defun ex-intr-addr (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (let ((DE0 (DQ-DE0 (MA-DQ MA))) (IFU (MA-IFU MA)) (pc (MA-pc MA)))
    (b-if (DE-valid? DE0) (DE-pc DE0)
    (b-if (IFU-valid? IFU) (IFU-pc IFU)
          pc))))

(defthm addr-ex-intr-addr
    (implies (MA-state-p MA) (addr-p (ex-intr-addr MA)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (MA-state-p MA) (integerp (ex-intr-addr MA))))))
(in-theory (disable ex-intr-addr))

; Return from the exception handling occurs in the current cycle.
; This is true when an RFEH instruction is committed.
(defun leave-excpt? (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (let ((ROBE (nth-ROBE (ROB-head (MA-ROB MA)) (MA-ROB MA))))
    (bs-and (ROBE-valid? ROBE)
            (ROBE-complete? ROBE)
            (ROBE-rfeh? ROBE)
            (b-not (LSU-pending-writes? (MA-LSU MA))))))

(defthm bitp-leave-excpt (bitp (leave-excpt? MA)))
(in-theory (disable leave-excpt?))

; If this line is raised, all entries in the pipeline are nullified.
; Flushing is usually taken when mispredicted branch, synchronization or
; an exception takes place.
(defun flush-all? (MA sigs)
  (declare (xargs :guard (and (MA-state-p MA) (MA-input-p sigs))))
  (bs-ior (commit-jmp? MA)
          (enter-excpt? MA)
```

```
          (ex-intr? MA sigs)
          (leave-excpt? MA)))

(defthm bitp-flush-all (bitp (flush-all? MA sigs)))
(in-theory (disable flush-all?))

; ROB-jmp-addr outputs the destination address of a ROB caused jump.
; This is valid when commit-jmp?, enter-excpt? or ex-intr? is raised.
; When leave-excpt? is raised, pc should get its new value from SR0.
; Notice that a conditional branch that is predicted to take place but
; actually does not also cause commit-jmp?.
(defun ROB-jmp-addr (ROB MA sigs)
  (declare (xargs :guard (and (ROB-p ROB) (MA-state-p MA) (MA-input-p sigs))))
  (let ((ROBE (nth-ROBE (ROB-head ROB) ROB)))
    (b-if (ex-intr? MA sigs) #x30
    (b-if (enter-excpt? MA)
          (logapp 4 0 (excpt-type (ROBE-excpt ROBE)))
    (b-if (bs-and (ROBE-valid? ROBE)
                  (ROBE-complete? ROBE)
                  (b-orc2 (ROBE-sync? ROBE)
                          (ROBE-br-actual? ROBE)))
          (addr (1+ (ROBE-pc ROBE)))
    (b-if (bs-and (ROBE-valid? ROBE)
                  (ROBE-complete? ROBE)
                  (ROBE-br-actual? ROBE))
          (addr (ROBE-val ROBE))
          0))))))

(defthm addr-p-rob-jmp-addr
    (implies (and (ROB-p ROB) (MA-state-p MA) (MA-input-p sigs))
             (addr-p (ROB-jmp-addr ROB MA sigs)))
  :hints (("goal" :in-theory (e/d (addr-p logapp* unsigned-byte-p* addr)
                                  (LOGAPP-0)))))
(in-theory (disable ROB-jmp-addr))

; ROB-write-reg? is raised when ROB writes its result into the register file.
(defun ROB-write-reg? (ROB)
  (declare (xargs :guard (ROB-p ROB)))
  (let ((ROBE (nth-ROBE (ROB-head ROB) ROB)))
    (bs-and (ROBE-valid? ROBE)
            (ROBE-complete? ROBE)
            (ROBE-wb? ROBE)
            (b-not (ROBE-wb-sreg? ROBE))
            (b-not (excpt-raised? (ROBE-excpt ROBE))))))

; ROB-write-sreg? is raised if ROB writes its result into the
; special register file.
(defun ROB-write-sreg? (ROB)
  (declare (xargs :guard (ROB-p ROB)))
  (let ((ROBE (nth-ROBE (ROB-head ROB) ROB)))
    (bs-and (ROBE-valid? ROBE)
            (ROBE-complete? ROBE)
            (ROBE-wb? ROBE)
            (ROBE-wb-sreg? ROBE)
            (b-not (excpt-raised? (ROBE-excpt ROBE))))))

; ROB-write-val returns the value to be written to the register file
; or the special register file.  When enter-excpt? is on, this bus is
; used to output the program counter value of the instruction that
; caused the raised exception or the next instruction.
(defun ROB-write-val (ROB MA)
  (declare (xargs :guard (and (ROB-p ROB) (MA-state-p MA))))
```

```
  (let ((robe (nth-ROBE (ROB-head ROB) ROB)))
    (b-if (enter-excpt? MA)
          (b-if (bv-eqv 2 (excpt-type (ROBE-excpt robe)) 0)
                (word (1+ (ROBE-pc robe)))
                (word (ROBE-pc robe)))
          (ROBE-val robe))))

; ROB-write-rid returns the register id to which the write-back value is
; written into.  If ROB-write-reg? is on, this designates a general register.
; If ROB-write-sreg? is on, this specifies a special register.  We assume
; that these lines are mutually exclusive.
(defun ROB-write-rid (ROB)
  (declare (xargs :guard (ROB-p ROB)))
  (ROBE-dest (nth-ROBE (ROB-head ROB) ROB)))

(defthm rname-p-rob-write-rid
    (implies (MA-state-p MA)
             (rname-p (rob-write-rid (MA-rob MA))))
  :hints (("goal" :in-theory (enable rob-write-rid))))

(deflabel begin-IU-output-def)
; select-IU-RS0? is set if IU-RS0 is chosen as the open reservation station
; slot for the new instruction to be dispatched.
(defun select-IU-RS0? (iu)
  (declare (xargs :guard (integer-unit-p iu)))
  (b-not (RS-valid? (IU-RS0 IU))))

(defthm bitp-select-IU-rs0 (bitp (select-IU-RS0? IU)))

(defun select-IU-RS1? (iu)
  (declare (xargs :guard (integer-unit-p iu)))
  (b-and (b-not (RS-valid? (IU-RS1 IU)))
         (RS-valid? (IU-RS0 IU))))

(defthm bitp-select-IU-rs1 (bitp (select-IU-RS1? IU)))

; If there is an available reservation station in IU, this line is raised.
(defun IU-ready? (iu)
  (declare (xargs :guard (integer-unit-p iu)))
  (b-nand (RS-valid? (IU-RS0 IU)) (RS-valid? (IU-RS1 IU))))

(defthm bitp-IU-ready? (bitp (IU-ready? IU)))

; The instruction at IU-RS0 is ready to be issued.
(defun IU-RS0-issue-ready? (iu)
  (declare (xargs :guard (integer-unit-p iu)))
  (let ((RS0 (IU-RS0 IU)))
    (bs-and (RS-valid? RS0)
            (RS-ready1? RS0)
            (b-ior (RS-op RS0) (RS-ready2? RS0)))))

(defthm bitp-IU-rs0-issue-ready? (bitp (IU-RS0-issue-ready? IU)))

; The instruction at IU-RS1 is ready to be issued.
(defun IU-RS1-issue-ready? (iu)
  (declare (xargs :guard (integer-unit-p iu)))
  (let ((RS1 (IU-RS1 IU)))
    (bs-and (RS-valid? RS1)
            (RS-ready1? RS1)
            (b-not (IU-RS0-issue-ready? IU))
            (b-ior (RS-op RS1) (RS-ready2? RS1)))))
```

```
(defthm bitp-IU-rs1-issue-ready? (bitp (IU-RS1-issue-ready? IU)))

; Tomasulo's tag output form the IU.
(defun IU-output-tag (IU)
  (declare (xargs :guard (integer-unit-p IU)))
  (b-if (IU-RS0-issue-ready? IU) (RS-tag (IU-RS0 IU))
        (b-if (IU-RS1-issue-ready? IU) (RS-tag (IU-RS1 IU))
              0)))

(defthm rob-index-p-IU-output-tag
    (implies (integer-unit-p IU)
             (rob-index-p (IU-output-tag IU))))

; The output value signal from the integer unit.
(defun IU-output-val (IU)
  (declare (xargs :guard (and (integer-unit-p IU))))
  (let ((RS0 (IU-RS0 IU)) (RS1 (IU-RS1 IU)))
    (b-if (IU-RS0-issue-ready? IU)
          (b-if (RS-op RS0)
                (RS-val1 RS0)
                (word (+ (RS-val1 RS0) (RS-val2 RS0))))
    (b-if (IU-RS1-issue-ready? IU)
          (b-if (RS-op RS1)
                (RS-val1 RS1)
                (word (+ (RS-val1 RS1) (RS-val2 RS1))))
          0))))
(deflabel end-IU-output-def)
(deftheory IU-output-def
    (definition-theory
        (set-difference-theories (universal-theory 'end-IU-output-def)
                                 (universal-theory 'begin-IU-output-def))))

(deflabel begin-MU-output-def)
; select-MU-RS0? is set if MU-RS0 is chosen as the open reservation station
; slot for the new instruction to be dispatched.
(defun select-MU-RS0? (MU)
  (declare (xargs :guard (mult-unit-p MU)))
  (b-not (RS-valid? (MU-RS0 MU))))

(defthm bitp-select-MU-RS0 (bitp (select-MU-RS0? MU)))

; MU-RS1 is selected as a holder for the MU dispatched instruction
(defun select-MU-RS1? (MU)
  (declare (xargs :guard (mult-unit-p MU)))
  (b-and (b-not (RS-valid? (MU-RS1 MU)))
         (RS-valid? (MU-RS0 MU))))

(defthm bitp-select-MU-RS1 (bitp (select-MU-RS1? MU)))

; IF there is an available reservation station in mult-unit,
; this line is raised.
(defun MU-ready? (MU)
  (declare (xargs :guard (mult-unit-p MU)))
  (b-nand (RS-valid? (MU-RS0 MU)) (RS-valid? (MU-RS1 MU))))

(defthm bitp-MU-ready (bitp (MU-ready? MU)))

; The instruction at MU-RS0 is ready to be issued.
(defun MU-RS0-issue-ready? (MU)
  (declare (xargs :guard (mult-unit-p MU)))
  (let ((RS0 (MU-RS0 MU)))
    (bs-and (RS-valid? RS0) (RS-ready1? RS0) (RS-ready2? RS0))))
```

```
(defthm bitp-MU-RS0-issue-ready (bitp (MU-RS0-issue-ready? MU)))

; The instruction at MU-RS1 is ready to be issued.
(defun MU-RS1-issue-ready? (MU)
  (declare (xargs :guard (mult-unit-p MU)))
  (let ((RS1 (MU-RS1 MU)))
    (bs-and (RS-valid? RS1) (RS-ready1? RS1) (RS-ready2? RS1)
            (b-not (MU-RS0-issue-ready? MU)))))

(defthm bitp-MU-RS1-issue-ready (bitp (MU-RS1-issue-ready? MU)))

;; We abstract away the implementation of the multiplier.
;; An actual multiplier is defined in multiplier-def.lisp.
;; We do not need the multiplier definition in the pipeline verification.
(encapsulate-impl multiplier
    ((ML1-output (ra rb) t)
     (ML2-output (data) t)
     (ML3-output (data) t))
((defun ML1-output (ra rb)
  (declare (xargs :guard (and (word-p ra) (word-p rb))))
  (cons ra rb))
 (defun ML2-output (data)
   (declare (xargs :guard t))
   data)
 (defun ML3-output (data)
   (declare (xargs :guard t))
   (if (consp data)
       (word (* (nfix (car data)) (nfix (cdr data))))
       0)))

((defthm ML-output-correct
    (implies (and (word-p ra) (word-p rb))
             (equal (ML3-output (ML2-output (ML1-output ra rb)))
                    (word (* ra rb)))))
(defthm ML3-output-correct
    (word-p (ML3-output data))))
)

; Alternative definition of the multiplier.
; We experimentally changed the definition of multiplier.  We plug in
; the Wallace tree multiplier here.
;  If you want to use a real multiplier implementation,
;  uncomment the following include-book command at the beginning of the file.
;      (include-book "multiplier-def")
;

;(encapsulate nil
;(local (include-book "multiplier-proof"))

;(defthm ML-output-correct
;    (implies (and (word-p ra) (word-p rb))
;             (equal (ML3-output (ML2-output (ML1-output ra rb)))
;                    (word (* ra rb)))))
;(defthm ML3-output-correct
;    (word-p (ML3-output data)))
;)
;(in-theory (disable ML1-output ML2-output ML3-output))

(deflabel end-MU-output-def)
(deftheory MU-output-def
    (definition-theory
```

```
        (set-difference-theories (universal-theory 'end-MU-output-def)
                                 (universal-theory 'begin-MU-output-def))))


(deflabel begin-bu-output-def)
; select-BU-RS0? is set if BU-RS0 is chosen as the open reservation station
; slot for the new instruction to go into.
(defun select-BU-RS0? (BU)
  (declare (xargs :guard (branch-unit-p BU)))
  (b-not (Bu-RS-valid? (BU-RS0 BU))))

(defthm bitp-select-BU-RS0? (bitp (select-BU-RS0? BU)))

; Choose RS1 of the BU as a possible slot for the dispatched instruction.
(defun select-BU-RS1? (BU)
  (declare (xargs :guard (branch-unit-p BU)))
  (b-and (b-not (BU-RS-valid? (BU-RS1 BU)))
         (BU-RS-valid? (BU-RS0 BU))))

(defthm bitp-select-BU-RS1? (bitp (select-BU-RS1? BU)))

; IF there is an available reservation station in BU-unit,
; this line is raised.
(defun BU-ready? (BU)
  (declare (xargs :guard (branch-unit-p BU)))
  (b-nand (BU-RS-valid? (BU-RS0 BU)) (BU-RS-valid? (BU-RS1 BU))))

(defthm bitp-BU-ready (bitp (BU-ready? BU)))

; The branch instruction at RS0 is ready to be issued.
(defun BU-RS0-issue-ready? (BU)
  (declare (xargs :guard (branch-unit-p BU)))
  (let ((RS0 (BU-RS0 BU)))
    (b-and (BU-RS-valid? RS0) (BU-RS-ready? RS0))))

(defthm bitp-BU-RS0-issue-ready (bitp (BU-RS0-issue-ready? BU)))

; The branch instruction at RS1 is ready to be issued.
(defun BU-RS1-issue-ready? (BU)
  (declare (xargs :guard (branch-unit-p BU)))
  (let ((RS1 (BU-RS1 BU)))
    (bs-and (BU-RS-valid? RS1) (BU-RS-ready? RS1)
            (b-not (BU-RS0-issue-ready? BU)))))

(defthm bitp-BU-RS1-issue-ready (bitp (BU-RS1-issue-ready? BU)))

; Tomasulo's tag output from the BU.  When an branch instruction
; is ready to complete, the tag for the instruction is put on the CDB.
(defun BU-output-tag (BU)
  (declare (xargs :guard (and (branch-unit-p BU))))
  (b-if (BU-RS0-issue-ready? BU) (BU-RS-tag (BU-RS0 BU))
        (b-if (BU-RS1-issue-ready? BU) (BU-RS-tag (BU-RS1 BU))
              0)))

(defthm rob-index-p-BU-output-tag
    (implies (branch-unit-p BU)
             (rob-index-p (BU-output-tag BU))))

; The output value from the BU is the branch outcome.  If the
; branch is taken, #xFFFF is output on the CDB.  0, otherwise.
(defun BU-output-val (BU)
  (declare (xargs :guard (branch-unit-p BU)))
```

```
  (b-if (BU-RS0-issue-ready? BU)
        (b-if (bv-eqv *word-size* (BU-RS-val (BU-RS0 BU)) 0)
              #xFFFF 0)
  (b-if (BU-RS1-issue-ready? BU)
        (b-if (bv-eqv *word-size* (BU-RS-val (BU-RS1 BU)) 0)
              #xFFFF 0)
        0)))

(deflabel end-bu-output-def)
(deftheory BU-output-def
    (definition-theory
        (set-difference-theories (universal-theory 'end-bu-output-def)
                                 (universal-theory 'begin-bu-output-def))))


(deflabel begin-LSU-output-def)
; select-LSU-RS0? is set if LSU-RS0 is chosen as the open reservation
; station slot for the new instruction to be dispatched.  This value is
; valid only if LSU-ready? is 1.
(defun select-LSU-RS0? (LSU)
  (declare (xargs :guard (load-store-unit-p LSU)))
  (b-if (LSU-rs1-head? LSU)
        (LSU-RS-valid? (LSU-RS1 LSU))
        (b-not (LSU-RS-valid? (LSU-RS0 LSU)))))

(defthm bitp-select-LSU-RS0?
    (implies (load-store-unit-p LSU) (bitp (select-LSU-RS0? LSU))))
(in-theory (disable select-LSU-RS0?))

; The negation of select-LSU-RS0?.
(defun select-LSU-RS1? (LSU)
  (declare (xargs :guard (load-store-unit-p LSU)))
  (b-if (LSU-rs1-head? LSU)
        (b-not (LSU-RS-valid? (LSU-RS1 LSU)))
        (LSU-RS-valid? (LSU-RS0 LSU))))

(defthm bitp-select-LSU-RS1?
    (implies (load-store-unit-p LSU) (bitp (select-LSU-RS1? LSU))))
(in-theory (disable select-LSU-RS1?))

; If there is an available reservation station in LSU-unit,
; this line is raised.
(defun LSU-ready? (LSU)
  (declare (xargs :guard (load-store-unit-p LSU)))
  (b-if (LSU-rs1-head? LSU)
        (b-not (LSU-RS-valid? (LSU-RS0 LSU)))
        (b-not (LSU-RS-valid? (LSU-RS1 LSU)))))

(defthm bitp-LSU-ready? (bitp (LSU-ready? LSU)))
(in-theory (disable LSU-ready?))

; LSU-RS0-issue-ready? is on when the load store instruction in RS0 is
; ready to be issued.
; RS0 should be a valid and necessary operands must be ready.  Also
; the order of instruction issues are important.  If RS1 contains a
; valid instruction and the next reservation flag points to RS1, it
; means that RS1 contains an earlier instruction than RS0.
(defun LSU-RS0-issue-ready? (LSU)
  (declare (xargs :guard (load-store-unit-p LSU)))
  (let ((RS0 (LSU-RS0 LSU))
        (RS1 (LSU-RS1 LSU)))
    (bs-and (LSU-RS-valid? RS0)
```

```
            (b-orc1 (LSU-RS-ld-st? RS0) (LSU-RS-rdy3? RS0))
            (LSU-RS-rdy1? RS0)
            (b-ior (LSU-RS-op RS0) (LSU-RS-rdy2? RS0))
            (b-nand (LSU-rs1-head? LSU) (LSU-RS-valid? RS1)))))

(defthm bitp-LSU-RS0-issue-ready? (bitp (LSU-RS0-issue-ready? LSU)))
(in-theory (disable LSU-RS0-issue-ready?))

; LSU-RS1-issue-ready? is on when the load store instruction in RS1 is ready
; to be issued.
(defun LSU-RS1-issue-ready? (LSU)
  (declare (xargs :guard (load-store-unit-p LSU)))
  (let ((RS0 (LSU-RS0 LSU))
        (RS1 (LSU-RS1 LSU)))
    (bs-and (LSU-RS-valid? RS1)
            (b-orc1 (LSU-RS-ld-st? RS1) (LSU-RS-rdy3? RS1))
            (LSU-RS-rdy1? RS1)
            (b-ior (LSU-RS-op RS1) (LSU-RS-rdy2? RS1))
            (b-orc2 (LSU-rs1-head? LSU) (LSU-RS-valid? RS0)))))

(defthm bitp-LSU-RS1-issue-ready? (bitp (LSU-RS1-issue-ready? LSU)))

; If the access addresses for the load instruction in a read buffer
; and the store instruction in a write buffer match, we can use the
; stored value as the result of the load instruction.
; LSU-address-match?  is on if there is a store instruction in the
; write buffer, which has the same access address.  We also check
; rbuf-wbuf0? and rbuf-wbuf1? fields to make sure the instructions in
; the write buffer are earlier instructions.
(defun LSU-address-match? (LSU)
  (declare (xargs :guard (load-store-unit-p LSU)))
  (let ((rbuf (LSU-rbuf LSU))
        (wbuf0 (LSU-wbuf0 LSU))
        (wbuf1 (LSU-wbuf1 LSU)))
    (b-ior (bs-and (bv-eqv *addr-size* (rbuf-addr rbuf) (wbuf-addr wbuf0))
                   (rbuf-valid? rbuf)
                   (rbuf-wbuf0? rbuf)
                   (wbuf-valid? wbuf0))
           (bs-and (bv-eqv *addr-size* (rbuf-addr rbuf) (wbuf-addr wbuf1))
                   (rbuf-valid? rbuf)
                   (rbuf-wbuf1? rbuf)
                   (wbuf-valid? wbuf1)))))

(defthm bitp-LSU-address-match (bitp (LSU-address-match? LSU)))
(in-theory (disable LSU-address-match?))

; If the addresses of load and store instruction in buffers match, we
; can use the value for the store instruction as the result of the
; load instruction. LSU-forward-wbuf associatively search the write
; buffer for the store instruction entry with a matching address, and
; returns the store value.  If there are more than one matching entry
; the value of the latest completed instruction is returned.
(defun LSU-forward-wbuf (LSU)
  (declare (xargs :guard (load-store-unit-p LSU)))
  (let ((rbuf (LSU-rbuf LSU))
        (wbuf0 (LSU-wbuf0 LSU))
        (wbuf1 (LSU-wbuf1 LSU)))
    (b-if (bs-and (bv-eqv *addr-size* (rbuf-addr rbuf) (wbuf-addr wbuf1))
                  (wbuf-valid? wbuf1)
                  (rbuf-wbuf1? rbuf))
          (wbuf-val wbuf1)
    (b-if (bs-and (bv-eqv *addr-size* (rbuf-addr rbuf) (wbuf-addr wbuf0))
```

```
                (wbuf-valid? wbuf0)
                (rbuf-wbuf0? rbuf))
          (wbuf-val wbuf0)
          0))))

; A memory stall occurs when the memory system does not respond for some
; reasons.  We don't model cache system or memory controller at all.
; Instead we give a very simplistic view of memory, which does or does
; not respond to the memory access request nondeterministicly.
; LSU-read-stall? can be set only when the LSU-rbuf1 is occupied.
; LSU-read-stall? does not indicate that the read address violates the
; memory protection, or there is an read-write address match with
; a write buffer entry.
(defun LSU-read-stall? (LSU sigs)
  (declare (xargs :guard (and (load-store-unit-p LSU) (MA-input-p sigs))))
  (b-andc2 (rbuf-valid? (LSU-rbuf LSU))
           (MA-input-data sigs)))

(defthm bitp-LSU-read-stall? (bitp (LSU-read-stall? LSU sigs)))
(in-theory (disable LSU-read-stall?))

; When the read address is prohibited by the memory protection mechanism,
; this line is raised.  If the machine is in the supervisor-mode, the
; error is not raised.
(defun LSU-read-prohibited? (LSU mem su)
  (declare (xargs :guard (and (load-store-unit-p LSU) (mem-p mem)
                              (bitp su))))
  (b-nor (readable-addr? (rbuf-addr (LSU-rbuf LSU)) mem) su))

(defthm bitp-LSU-read-prohibited? (bitp (LSU-read-prohibited? LSU mem su)))
(in-theory (disable LSU-read-prohibited?))


; If the write buffer entry at wbuf0 is valid, but not yet completed,
; we may be ready to check its address and complete the instruction.
; The actual memory access takes place after the store instruction commits.
; check-wbuf0? is on when the entry is actually sent to next latch
; to complete.
(defun check-wbuf0? (LSU)
  (declare (xargs :guard (load-store-unit-p LSU)))
  (bs-and (wbuf-valid? (LSU-wbuf0 LSU))
          (b-not (wbuf-complete? (LSU-wbuf0 LSU)))))

(defthm bitp-check-wbuf0? (bitp (check-wbuf0? LSU)))
(in-theory (disable check-wbuf0?))

(defun check-wbuf1? (LSU)
  (declare (xargs :guard (load-store-unit-p LSU)))
  (bs-and (wbuf-valid? (LSU-wbuf1 LSU))
          (b-not (wbuf-complete? (LSU-wbuf1 LSU)))
          (b-not (check-wbuf0? LSU))))

(defthm bitp-check-wbuf1? (bitp (check-wbuf1? LSU)))
(in-theory (disable check-wbuf1?))

; We release the read operation, if no memory permission check is done
; for the write operations during this cycle, and read operation can
; be done during this cycle.  Memory read operation can be completed
; during this cycle if either one of write operation has the same access
; address as the read operation, memory protection is violated by the
; read operation, or memory read can be carried out without stall.
(defun release-rbuf? (LSU MA sigs)
```

```
  (declare (xargs :guard (and (load-store-unit-p LSU) (MA-state-p MA)
                              (MA-input-p sigs))))
  (bs-and (rbuf-valid? (LSU-rbuf LSU))
          (b-not (check-wbuf0? LSU))
          (b-not (check-wbuf1? LSU))
          (bs-ior (LSU-address-match? LSU)
                  (LSU-read-prohibited? LSU (MA-mem MA) (SRF-su (MA-SRF MA)))
                  (b-not (LSU-read-stall? LSU sigs)))))
(defthm bitp-release-rbuf?
    (bitp (release-rbuf? LSU MA sigs)))
(in-theory (disable release-rbuf?))

; If the store instruction at the head of the write buffer has committed
; and we are ready to write the value to the memory, release-wbuf0-ready?
; is set.
(defun release-wbuf0-ready? (LSU)
  (declare (xargs :guard (load-store-unit-p LSU)))
  (let ((wbuf0 (LSU-wbuf0 LSU)))
    (b-and (wbuf-valid? wbuf0) (wbuf-commit? wbuf0))))

(defthm bitp-release-wbuf0-ready? (bitp (release-wbuf0-ready? LSU)))
(in-theory (disable release-wbuf0-ready?))

; LSU-write-stall? is set when a write buffer entry tries to write its result
; to the memory and the memory does not respond.  It is only set when
; Release-wbuf0-ready? is on.
(defun LSU-write-stall? (LSU sigs)
  (declare (xargs :guard (and (load-store-unit-p LSU) (MA-input-p sigs))))
  (b-andc2 (release-wbuf0-ready? LSU) (MA-input-data sigs)))

(defthm bitp-LSU-write-stall? (bitp (LSU-write-stall? LSU sigs)))
(in-theory (disable LSU-write-stall?))

; Write operation is performed and the instruction is released.
; Memory operations are done in this priority.
;  1 Write operation memory protection check
;  2 Read operation
;  3 Write operation.
; Thus write operation is done only when no read operation is pending,
; and the no protection check for any write operation is pending.
; The write operation memory check has a higher priority than
; the read operation.  Using the load-bypassing, a read operation
; may steal a value from a write operation, that may in the future cause
; an exception.
(defun release-wbuf0? (LSU sigs)
  (declare (xargs :guard (and (load-store-unit-p LSU) (MA-input-p sigs))))
  (bs-and (release-wbuf0-ready? LSU)
          (b-not (LSU-write-stall? LSU sigs))
          (b-not (rbuf-valid? (LSU-rbuf LSU)))
          (b-not (check-wbuf1? LSU))))

(defthm bitp-release-wbuf0? (bitp (release-wbuf0? LSU sigs)))
(in-theory (disable release-wbuf0?))

; If the memory access is prohibited by the memory system,
; LSU-write-prohibited? is raised.  This is valid only when
; check-wbuf0? or check-wbuf1? is on.  The instruction in the corresponding
; write buffer entry causes the write address violation.
(defun LSU-write-prohibited? (LSU mem su)
  (declare (xargs :guard (and (load-store-unit-p LSU) (mem-p mem) (bitp su))))
  (b-if (check-wbuf0? LSU)
        (b-nor (writable-addr? (wbuf-addr (LSU-wbuf0 LSU)) mem) su)
```

```
  (b-if (check-wbuf1? LSU)
        (b-nor (writable-addr? (wbuf-addr (LSU-wbuf1 LSU)) mem) su)
        0)))

(defthm bitp-LSU-write-prohibited? (bitp (LSU-write-prohibited? LSU mem su)))
(in-theory (disable LSU-write-prohibited?))
(deflabel end-LSU-output-def)
(deftheory LSU-output-def
    (definition-theory
        (set-difference-theories (universal-theory 'end-LSU-output-def)
                                 (universal-theory 'begin-LSU-output-def))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Dispatching of instructions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Dispatching an instruction that completes immediately.
(defun dispatch-no-unit? (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (bs-and (DQ-ready-no-unit? (MA-DQ MA))
          (b-not (ROB-full? (MA-ROB MA)))
          (b-not (ROB-exintr? (MA-ROB MA)))))

(defthm bitp-dispatch-no-unit? (bitp (dispatch-no-unit? MA)))
(in-theory (disable dispatch-no-unit?))

; Dispatching an instruction to the IU.
(defun dispatch-to-IU? (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (bs-and (DQ-ready-to-IU? (MA-DQ MA))
          (b-not (ROB-full? (MA-ROB MA)))
          (b-not (ROB-exintr? (MA-ROB MA)))
          (IU-ready? (MA-IU MA))))

(defthm bitp-dispatch-to-IU (bitp (dispatch-to-IU? MA)))
(in-theory (disable dispatch-to-IU?))

; Dispatching an instruction to the MU.
(defun dispatch-to-MU? (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (bs-and (DQ-ready-to-MU? (MA-DQ MA))
          (b-not (ROB-full? (MA-ROB MA)))
          (b-not (ROB-exintr? (MA-ROB MA)))
          (MU-ready? (MA-MU MA))))

(defthm bitp-dispatch-to-MU (bitp (dispatch-to-MU? MA)))
(in-theory (disable dispatch-to-MU?))

; Dispatching an instruction to the LSU.
(defun dispatch-to-LSU? (MA)
  (declare (xargs :guard  (MA-state-p MA)))
  (bs-and (DQ-ready-to-LSU? (MA-DQ MA))
          (b-not (ROB-full? (MA-ROB MA)))
          (b-not (ROB-exintr? (MA-ROB MA)))
          (LSU-ready? (MA-LSU MA))))

(defthm bitp-dispatch-to-LSU? (bitp (dispatch-to-LSU? MA)))
(in-theory (disable dispatch-to-LSU?))

; Dispatching an instruction to the BU.
(defun dispatch-to-BU? (MA)
  (declare (xargs :guard (MA-state-p MA)))
```

```
  (bs-and (DQ-ready-to-BU? (MA-DQ MA))
          (b-not (ROB-full? (MA-ROB MA)))
          (b-not (ROB-exintr? (MA-ROB MA)))
          (BU-ready? (MA-BU MA))))

(defthm bitp-dispatch-to-BU (bitp (dispatch-to-BU? MA)))
(in-theory (disable dispatch-to-BU?))

; This line is raised if any instruction is ready to dispatch.
; A dispatch actually takes place if the corresponding execution
; unit has an empty reservation station.
(defun dispatch-inst? (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (bs-ior (dispatch-no-unit? MA)
          (dispatch-to-IU? MA)
          (dispatch-to-MU? MA)
          (dispatch-to-LSU? MA)
          (dispatch-to-BU? MA)))

(defthm bitp-dispatch-inst? (bitp (dispatch-inst? MA)))
(in-theory (disable dispatch-inst?))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Common Data Bus Arbitration
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deflabel begin-CDB-arb-def)

; The CDB is ready for the output from the LSU.
(defun CDB-for-LSU? (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (LSU-latch-valid? (LSU-lch (MA-LSU MA))))

; The CDB is ready for the output from the MU.
(defun CDB-for-MU? (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (bs-and (b-not (CDB-for-LSU? MA))
          (MU-latch2-valid? (MU-lch2 (MA-MU MA)))))

; The CDB is ready for the output from the BU.
(defun CDB-for-BU? (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (bs-and (b-not (CDB-for-LSU? MA))
          (b-not (CDB-for-MU? MA))
          (b-ior (BU-RS0-issue-ready? (MA-BU MA))
                 (BU-RS1-issue-ready? (MA-BU MA)))))

; The CDB is ready for the output from the IU.
(defun CDB-for-IU? (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (bs-and (b-not (CDB-for-LSU? MA))
          (b-not (CDB-for-BU? MA))
          (b-not (CDB-for-MU? MA))
          (b-ior (IU-RS0-issue-ready? (MA-IU MA))
                 (IU-RS1-issue-ready? (MA-IU MA)))))
(deflabel end-CDB-arb-def)

(defthm bitp-CDB-for-LSU  (implies (MA-state-p MA) (bitp (CDB-for-LSU? MA))))
(defthm bitp-CDB-for-MU  (bitp (CDB-for-MU? MA)))
(defthm bitp-CDB-for-BU  (bitp (CDB-for-BU? MA)))
(defthm bitp-CDB-for-IU  (bitp (CDB-for-IU? MA)))

(deftheory CDB-arb-def
```

```
    (set-difference-theories (universal-theory 'end-CDB-arb-def)
                             (universal-theory 'begin-CDB-arb-def)))
(in-theory (disable CDB-arb-def))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Definition of the issues of instructions at each execution unit.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; issue-IU-RS0 and issue-IU-RS1 are set if the instruction in the
; corresponding reservation station is issued to the execution unit.
; Otherwise, cleared.  Notice issue-IU-RS0? is on, RS-valid? for RS0 is on.
; Similarly with RS1.
(deflabel begin-issue-logic-def)
(defun issue-IU-RS0? (iu MA)
  (declare (xargs :guard (and (integer-unit-p iu) (MA-state-p MA))))
  (b-and (IU-RS0-issue-ready? IU) (CDB-for-IU? MA)))

(defthm bitp-issue-IU-RS0  (bitp (issue-IU-RS0? IU MA)))

(defun issue-IU-RS1? (iu MA)
  (declare (xargs :guard (and (integer-unit-p iu) (MA-state-p MA))))
  (bs-and (IU-RS1-issue-ready? IU) (CDB-for-IU? MA)))

(defthm bitp-issue-IU-RS1 (bitp (issue-IU-RS1? IU MA)))

(defun issue-MU-RS0? (MU MA)
  (declare (xargs :guard (and (mult-unit-p MU) (MA-state-p MA))))
  (b-and (MU-RS0-issue-ready? MU)
         (bs-ior (CDB-for-MU? MA)
                 (b-not (MU-latch1-valid? (MU-lch1 MU)))
                 (b-not (MU-latch2-valid? (MU-lch2 MU))))))

(defthm bitp-issue-MU-RS0 (bitp (issue-MU-RS0? MU MA)))

(defun issue-MU-RS1? (MU MA)
  (declare (xargs :guard (and (mult-unit-p MU) (MA-state-p MA))))
  (bs-and (MU-RS1-issue-ready? MU)
          (bs-ior (CDB-for-MU? MA)
                  (b-not (MU-latch1-valid? (MU-lch1 MU)))
                  (b-not (MU-latch2-valid? (MU-lch2 MU))))))

(defthm bitp-issue-MU-RS1 (bitp (issue-MU-RS1? MU MA)))

(defun issue-BU-RS0? (BU MA)
  (declare (xargs :guard (and (branch-unit-p BU) (MA-state-p MA))))
  (b-and (BU-RS0-issue-ready? BU) (CDB-for-BU? MA)))

(defthm bitp-issue-BU-RS0? (bitp (issue-BU-RS0? BU MA)))

(defun issue-BU-RS1? (BU MA)
  (declare (xargs :guard (and (branch-unit-p BU) (MA-state-p MA))))
  (b-and (BU-RS1-issue-ready? BU) (CDB-for-BU? MA)))

(defthm bitp-issue-BU-RS1 (bitp (issue-BU-RS1? BU MA)))

(defun issue-LSU-RS0? (LSU MA sigs)
  (declare (xargs :guard (and (load-store-unit-p LSU) (MA-state-p MA)
                              (MA-input-p sigs))))
  (bs-and (LSU-RS0-issue-ready? LSU)
          (b-if (LSU-RS-ld-st? (LSU-RS0 LSU))
                (b-ior (b-not (wbuf-valid? (LSU-wbuf1 LSU)))
                       (release-wbuf0? LSU sigs))
                (bs-ior (b-not (rbuf-valid? (LSU-rbuf LSU)))
```

```
                        (release-rbuf? LSU MA sigs)))))

(defthm bitp-issue-LSU-RS0? (bitp (issue-LSU-RS0? LSU MA sigs)))
(in-theory (disable issue-LSU-RS0?))

(defun issue-LSU-RS1? (LSU MA sigs)
  (declare (xargs :guard (and (load-store-unit-p LSU)
                              (MA-state-p MA) (MA-input-p sigs))))
  (bs-and (LSU-RS1-issue-ready? LSU)
          (b-if (LSU-RS-ld-st? (LSU-RS1 LSU))
                (b-ior (b-not (wbuf-valid? (LSU-wbuf1 LSU)))
                       (release-wbuf0? LSU sigs))
                (bs-ior (b-not (rbuf-valid? (LSU-rbuf LSU)))
                        (release-rbuf? LSU MA sigs)))))

(defthm bitp-issue-LSU-RS1? (bitp (issue-LSU-RS1? LSU MA sigs)))
(in-theory (disable issue-LSU-RS1?))

(deflabel end-issue-logic-def)

(deftheory issue-logic-def
    (definition-theory
        (set-difference-theories (universal-theory 'end-issue-logic-def)
                                 (universal-theory 'begin-issue-logic-def))))

(deflabel begin-CDB-out-def)
; Common Data Bus should raise ready? flag when a datum is ready.
; CDB-tag posts the destination ROB entry index (or what we call Tomasulo's
; tag), while CDB-val is the result of the executed instruction.
(defun CDB-ready? (MA)
  (declare (xargs :guard  (MA-state-p MA)))
  (bs-ior (LSU-latch-valid? (LSU-lch (MA-LSU MA)))
          (MU-latch2-valid? (MU-lch2 (MA-MU MA)))
          (BU-RS0-issue-ready? (MA-BU MA))
          (BU-RS1-issue-ready? (MA-BU MA))
          (IU-RS0-issue-ready? (MA-IU MA))
          (IU-RS1-issue-ready? (MA-IU MA))))

(defthm bitp-CDB-ready   (bitp (CDB-ready? MA)))
(in-theory (disable CDB-ready?))

; Tomasulo's tag, i.e., the index to the ROB entry to which the completing
; instruction is assigned to is put on the tag signal of the CDB.
(defun CDB-tag (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (b-if (CDB-for-IU? MA)
        (IU-output-tag (MA-IU MA))
  (b-if (CDB-for-BU? MA)
        (BU-output-tag (MA-BU MA))
  (b-if (CDB-for-MU? MA)
        (MU-latch2-tag (MU-lch2 (MA-MU MA)))
  (b-if (CDB-for-LSU? MA)
        (LSU-latch-tag (LSU-lch (MA-LSU MA)))
        0)))))

(defthm rob-index-p-CDB-tag
    (implies (MA-state-p MA) (rob-index-p (CDB-tag MA)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (MA-state-p MA) (integerp (CDB-tag MA))))))
```

```
(in-theory (disable CDB-tag))

; The exception flag for the completing instruction is put on the CDB.
(defun CDB-excpt (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (b-if (CDB-for-LSU? MA) (LSU-latch-excpt (LSU-lch (MA-LSU MA))) 0))

(defthm excpt-flags-p-CDB-excpt
    (implies (MA-state-p MA) (excpt-flags-p (CDB-excpt MA))))
(in-theory (disable CDB-excpt))

; The result of the completing instruction is put on the val signal of the
; CDB.  For branch instructions, see BU-output-val.
(defun CDB-val (MA)
  (declare (xargs :guard  (MA-state-p MA)))
  (b-if (CDB-for-IU? MA)
        (IU-output-val (MA-IU MA))
  (b-if (CDB-for-BU? MA)
        (BU-output-val (MA-BU MA))
  (b-if (CDB-for-MU? MA)
        (ML3-output (MU-latch2-data (MU-lch2 (MA-MU MA))))
  (b-if (CDB-for-LSU? MA)
        (LSU-latch-val (LSU-lch (MA-LSU MA)))
        0)))))

(defthm word-p-CDB-val
    (implies (MA-state-p MA) (word-p (CDB-val MA)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (MA-state-p MA) (integerp (CDB-val MA)))
             :hints (("Goal" :in-theory (enable word-p))))))
(in-theory (disable CDB-val))

; The CDB is ready for the instruction which is assigned to the ROB entry
; idx.
(defun CDB-ready-for? (idx MA)
  (declare (xargs :guard (and (rob-index-p idx) (MA-state-p MA))))
  (b-and (CDB-ready? MA)
         (bv-eqv *rob-index-size* idx (CDB-tag MA))))

(defthm bitp-CDB-ready-for (bitp (CDB-ready-for? idx MA)))
(in-theory (disable CDB-ready-for?))

(deflabel end-CDB-out-def)
(deftheory CDB-out-def
    (definition-theory
        (set-difference-theories (universal-theory 'end-CDB-out-def)
                                 (universal-theory 'begin-CDB-out-def))))

(deftheory CDB-def
    (union-theories (theory 'CDB-out-def)
                    (theory 'CDB-arb-def)))

; The control vector of the dispatched instruction.
(defun dispatch-cntlv (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (DE-cntlv (DQ-DEO (MA-DQ MA))))

(defthm cntlv-p-dispatch-cntlv
    (implies (MA-state-p MA) (cntlv-p (dispatch-cntlv MA)))
  :rule-classes
```

```
  ((:rewrite)
   (:rewrite :corollary
             (implies (MA-state-p MA) (integerp (dispatch-cntlv MA)))
             :hints (("Goal" :in-theory (enable cntlv-p))))))
(in-theory (disable dispatch-cntlv))

; Destination register of the dispatched instruction.
(defun dispatch-dest-reg (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (DQ-out-dest-reg (MA-DQ MA)))

(defthm rname-p-dispatch-dest-reg
    (implies (MA-state-p MA) (rname-p (dispatch-dest-reg MA)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (MA-state-p MA) (integerp (dispatch-dest-reg MA)))
             :hints (("Goal" :in-theory (enable rname-p))))))
(in-theory (disable dispatch-dest-reg))

; The exception condition of the dispatched instruction.
(defun dispatch-excpt (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (DE-excpt (DQ-DEO (MA-DQ MA))))

(defthm excpt-flags-p-dispatch-excpt
    (implies (MA-state-p MA) (excpt-flags-p (dispatch-excpt MA))))
(in-theory (disable dispatch-excpt))

; The program counter of the dispatched instruction.
(defun dispatch-pc (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (DE-pc (DQ-DEO (MA-DQ MA))))

(defthm addr-p-dispatch-pc
    (implies (MA-state-p MA) (addr-p (dispatch-pc MA))))
(in-theory (disable dispatch-pc))

; Source value 1 is ready for the dispatched instruction.
(defun dispatch-ready1? (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (bs-ior (DQ-out-ready1? (MA-DQ MA))
          (ROBE-complete? (nth-ROBE (DQ-out-tag1 (MA-DQ MA)) (MA-ROB MA)))
          (CDB-ready-for? (DQ-out-tag1 (MA-DQ MA)) MA)))

(defthm bitp-dispatch-ready1? (bitp (dispatch-ready1? MA)))
(in-theory (disable dispatch-ready1?))

; The source operand for a dispatched instruction come from three possible
; sources: the register file, ROB and CDB.  If DQ-out-ready2 line is
; on, the value stored in the register file is the correct value.  If
; the complete? flag of the corresponding ROB entry is on, the correct
; value is redirected from the ROB entry.  If the CDB outputs the result
; of an instruction on which the dispatched instruction truly
; depends, the value on CDB is read.  Otherwise, the source operand is not
; ready.
(defun dispatch-val1 (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (b-if (DQ-out-ready1? (MA-DQ MA))
        (DQ-read-val1 (MA-DQ MA) MA)
        (b-if (ROBE-complete? (nth-ROBE (DQ-out-tag1 (MA-DQ MA)) (MA-ROB MA)))
              (ROBE-val (nth-ROBE (DQ-out-tag1 (MA-DQ MA)) (MA-ROB MA)))
```

```
              (b-if (CDB-ready-for? (DQ-out-tag1 (MA-DQ MA)) MA)
                    (CDB-val MA)
                    0))))

(defthm word-p-dispatch-val1
    (implies (MA-state-p MA) (word-p (dispatch-val1 MA))))
(in-theory (disable dispatch-val1))

;  Tomasulo's tag for the producer instruction of the source operand 1.
(defun dispatch-src1 (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (DQ-out-tag1 (MA-DQ MA)))

(defthm rob-index-p-dispatch-src1
    (implies (MA-state-p MA) (rob-index-p (dispatch-src1 MA))))
(in-theory (disable dispatch-src1))

; Source operand 2 is ready.
(defun dispatch-ready2? (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (bs-ior (DQ-out-ready2? (MA-DQ MA))
          (ROBE-complete? (nth-ROBE (DQ-out-tag2 (MA-DQ MA)) (MA-ROB MA)))
          (CDB-ready-for? (DQ-out-tag2 (MA-DQ MA)) MA)))

(defthm bitp-dispatch-ready2  (bitp (dispatch-ready2? MA)))
(in-theory (disable dispatch-ready2?))

; The value for a dispatched instruction come from three possible sources:
; the register file, ROB and CDB.  If DQ-out-ready2 line is on, the value
; stored in the register file is the correct value.  Otherwise, we have to
; get the value from the ROB entry or the CDB.  See dispatch-val1.
(defun dispatch-val2 (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (b-if (DQ-out-ready2? (MA-DQ MA))
        (read-reg (DQ-out-reg2 (MA-DQ MA)) (MA-RF MA))
        (b-if (ROBE-complete? (nth-ROBE (DQ-out-tag2 (MA-DQ MA)) (MA-ROB MA)))
              (ROBE-val (nth-ROBE (DQ-out-tag2 (MA-DQ MA)) (MA-ROB MA)))
              (b-if (CDB-ready-for? (DQ-out-tag2 (MA-DQ MA)) MA)
                    (CDB-val MA)
                    0))))

(defthm word-p-dispatch-val2
    (implies (MA-state-p MA) (word-p (dispatch-val2 MA))))
(in-theory (disable dispatch-val2))

; Tomasulo's tag for the producer instruction for the source operand 2.
(defun dispatch-src2 (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (DQ-out-tag2 (MA-DQ MA)))

(defthm rob-index-p-dispatch-src2
    (implies (MA-state-p MA) (rob-index-p (dispatch-src2 MA))))
(in-theory (disable dispatch-src2))

; Source operand 3 is ready.
(defun dispatch-ready3? (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (bs-ior (DQ-out-ready3? (MA-DQ MA))
          (ROBE-complete? (nth-ROBE (DQ-out-tag3 (MA-DQ MA)) (MA-ROB MA)))
          (CDB-ready-for? (DQ-out-tag3 (MA-DQ MA)) MA)))

(defthm bitp-dispatch-ready3 (bitp (dispatch-ready3? MA)))
```

```
(in-theory (disable dispatch-ready3?))

; Source operand 3 for the dispatched instruction, if ready.  See
; dispatch-val1.
(defun dispatch-val3 (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (b-if (DQ-out-ready3? (MA-DQ MA))
        (read-reg (DQ-out-reg3 (MA-DQ MA)) (MA-RF MA))
        (b-if (ROBE-complete? (nth-ROBE (DQ-out-tag3 (MA-DQ MA)) (MA-ROB MA)))
              (ROBE-val (nth-ROBE (DQ-out-tag3 (MA-DQ MA)) (MA-ROB MA)))
        (b-if (CDB-ready-for? (DQ-out-tag3 (MA-DQ MA)) MA)
              (CDB-val MA)
             0))))

(defthm word-p-dispatch-val3
    (implies (MA-state-p MA) (word-p (dispatch-val3 MA))))
(in-theory (disable dispatch-val3))

; Tomasulo's tag for the producer instruction for the source operand 3.
(defun dispatch-src3 (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (DQ-out-tag3 (MA-DQ MA)))

(defthm rob-index-p-dispatch-src3
    (implies (MA-state-p MA) (rob-index-p (dispatch-src3 MA))))
(in-theory (disable dispatch-src3))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Step functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deflabel begin-MA-step-functions)
; The next state of pc.
(defun step-pc (MA sigs)
  (declare (xargs :guard (and (MA-state-p MA) (MA-input-p sigs))))
  (b-if (b-ior (enter-excpt? MA)
               (ex-intr? MA sigs))
        (rob-jmp-addr (MA-ROB MA) MA sigs)
  (b-if (leave-excpt? MA) (addr (SRF-sr0 (MA-SRF MA)))
  (b-if (commit-jmp? MA) (rob-jmp-addr (MA-ROB MA) MA sigs)
  (b-if (b-andc2 (IFU-branch-predict? (MA-IFU MA) MA sigs)
                 (DQ-full? (MA-DQ MA)))
        (IFU-branch-target (MA-IFU MA))
  (b-if (b-and (b-nand (IFU-valid? (MA-IFU MA)) (DQ-full? (MA-DQ MA)))
               (MA-input-fetch sigs))
        (addr (1+ (MA-pc MA)))
        (MA-pc MA)))))))

(defthm addr-p-step-pc
    (implies (and (MA-state-p MA) (MA-input-p sigs))
             (addr-p (step-pc MA sigs))))
(in-theory (disable step-pc))

; The next state of register file.
(defun step-RF (MA)
  (declare (xargs :guard (and (MA-state-p MA))))
  (b-if (ROB-write-reg? (MA-ROB MA))
        (write-reg (ROB-write-val (MA-ROB MA) MA)
                   (ROB-write-rid (MA-ROB MA))
                   (MA-RF MA))
        (MA-RF MA)))

(defthm RF-p-step-RF
    (implies (MA-state-p MA)
```

```
            (RF-p (step-RF MA))))
(in-theory (disable step-RF))

; The next state of the special register.
(defun step-SRF (MA sigs)
  (declare (xargs :guard (and (MA-state-p MA) (MA-input-p sigs))))
  (b-if (ex-intr? MA sigs)
        (SRF 1 (word (ex-intr-addr MA)) (word (SRF-su (MA-SRF MA))))
  (b-if (enter-excpt? MA)
        (SRF 1 (ROB-write-val (MA-ROB MA) MA)
               (word (SRF-su (MA-SRF MA))))
  (b-if (leave-excpt? MA)
        (update-SRF (MA-SRF MA) :su (logcar (SRF-sr1 (MA-SRF MA))))
  (b-if (ROB-write-sreg? (MA-ROB MA))
        (write-sreg (ROB-write-val (MA-ROB MA) MA)
                    (ROB-write-rid (MA-ROB MA))
                    (MA-SRF MA))
        (MA-SRF MA))))))

(defthm SRF-p-step-SRF
    (implies (and (MA-state-p MA) (MA-input-p sigs))
             (SRF-p (step-SRF MA sigs))))
(in-theory (disable step-SRF))

; The next state of the IFU.
(defun step-IFU (MA sigs)
  (declare (xargs :guard (and (MA-state-p MA) (MA-input-p sigs))))
  ; If the IFU and DQ are full, the instruction at IFU stalls.
  (b-if (b-and (IFU-valid? (MA-IFU MA)) (DQ-full? (MA-DQ MA)))
        (update-IFU (MA-IFU MA)
                    :valid? (b-not (flush-all? MA sigs)))
  ; If instruction fetch error occurs, an instruction word is 0.
  ; This instruction will eventually raise a fetch error exception.
  (b-if (IFU-fetch-prohibited? (MA-pc MA) (MA-mem MA) (SRF-su (MA-SRF MA)))
        (IFU (b-nor (IFU-branch-predict? (MA-IFU MA) MA sigs)
                    (flush-all? MA sigs))
             #b101 (MA-pc MA) 0)
  ; If the memory does respond, we fetch an instruction.
  (b-if (MA-input-fetch sigs)
        (IFU (b-nor (IFU-branch-predict? (MA-IFU MA) MA sigs)
                    (flush-all? MA sigs))
             0 (MA-pc MA) (read-mem (MA-pc MA) (MA-mem MA)))
  ; If no memory response returns, we don't fetch an instruction.
        (IFU 0 0 0 0)))))

(defthm IFU-p-step-IFU
    (implies (and (MA-state-p MA) (MA-input-p sigs))
             (IFU-p (step-IFU MA sigs))))
(in-theory (disable step-IFU))

;  The definition of step-DQ.
;   The dispatch queue is a FIFO buffer.  Unlike the ROB, we tried to
;   implemented like a systolic buffer.  (I am not sure whether this
;   word applies.)  An entry passes to the next entry when an instruction
;   is dispatched.  For instance, the content of DE3 is passed to DE2,
;   the content of DE2 is passed to DE1.  DE?-out is the value passed to the
;   next entry.
(defun DE3-out (DQ MA sigs)
  (declare (xargs :guard (and (DQ-p DQ) (MA-state-p MA) (MA-input-p sigs))))
  (b-if (DE-valid? (DQ-DE3 DQ))
        (DQ-DE3 DQ)
        (decode-output (MA-IFU MA) MA sigs)))
```

```
(defthm dispatch-entry-p-DE3-out
    (implies (and (DQ-p DQ) (MA-state-p MA) (MA-input-p sigs))
             (dispatch-entry-p (DE3-out DQ MA sigs))))

(in-theory (disable DE3-out))

(defun DE2-out (DQ MA sigs)
  (declare (xargs :guard (and (DQ-p DQ) (MA-state-p MA) (MA-input-p sigs))))
  (b-if (DE-valid? (DQ-DE2 DQ))
        (DQ-DE2 DQ)
        (DE3-out DQ MA sigs)))

(defthm dispatch-entry-p-DE2-out
    (implies (and (DQ-p DQ) (MA-state-p MA) (MA-input-p sigs))
             (dispatch-entry-p (DE2-out DQ MA sigs))))

(in-theory (disable DE2-out))

(defun DE1-out (DQ MA sigs)
  (declare (xargs :guard (and (DQ-p DQ) (MA-state-p MA) (MA-input-p sigs))))
  (b-if (DE-valid? (DQ-DE1 DQ))
        (DQ-DE1 DQ)
        (DE2-out DQ MA sigs)))

(defthm dispatch-entry-p-DE1-out
    (implies (and (DQ-p DQ) (MA-state-p MA) (MA-input-p sigs))
             (dispatch-entry-p (DE1-out DQ MA sigs))))

(in-theory (disable DE1-out))

; The next state of the DE0.
(defun step-DE0 (DQ MA sigs)
  (declare (xargs :guard (and (DQ-p DQ) (MA-state-p MA) (MA-input-p sigs))))
  (b-if (b-orc1 (DE-valid? (DQ-DE0 DQ)) (dispatch-inst? MA))
        (update-dispatch-entry (DE1-out DQ MA sigs)
                   :valid? (b-andc2 (DE-valid? (DE1-out DQ MA sigs))
                                    (flush-all? MA sigs)))
        (update-dispatch-entry (DQ-DE0 DQ)
                   :valid? (b-andc2 (DE-valid? (DQ-DE0 DQ))
                                    (flush-all? MA sigs)))))

; The next state of the DE1.
(defun step-DE1 (DQ MA sigs)
  (declare (xargs :guard (and (DQ-p DQ) (MA-state-p MA) (MA-input-p sigs))))
  (b-if (b-ior (b-and (DE-valid? (DQ-DE1 DQ)) (dispatch-inst? MA))
               (bs-and (b-not (DE-valid? (DQ-DE1 DQ)))
                       (DE-valid? (DQ-DE0 DQ))
                       (b-not (dispatch-inst? MA))))
        (update-dispatch-entry (DE2-out DQ MA sigs)
                   :valid? (b-andc2 (DE-valid? (DE2-out DQ MA sigs))
                                    (flush-all? MA sigs)))
        (update-dispatch-entry (DQ-DE1 DQ)
                   :valid? (b-andc2 (DE-valid? (DQ-DE1 DQ))
                                    (flush-all? MA sigs)))))

; The next state of the DE2.
(defun step-DE2 (DQ MA sigs)
  (declare (xargs :guard (and (DQ-p DQ) (MA-state-p MA) (MA-input-p sigs))))
  (b-if (b-ior (b-and (DE-valid? (DQ-DE2 DQ)) (dispatch-inst? MA))
               (bs-and (b-not (DE-valid? (DQ-DE2 DQ)))
                       (DE-valid? (DQ-DE1 DQ))
```

```
                    (b-not (dispatch-inst? MA))))
       (update-dispatch-entry (DE3-out DQ MA sigs)
                 :valid? (b-andc2 (DE-valid? (DE3-out DQ MA sigs))
                                  (flush-all? MA sigs)))
       (update-dispatch-entry (DQ-DE2 DQ)
                 :valid? (b-andc2 (DE-valid? (DQ-DE2 DQ))
                                  (flush-all? MA sigs)))))

; The next state of the DE3.
(defun step-DE3 (DQ MA sigs)
  (declare (xargs :guard (and (DQ-p DQ) (MA-state-p MA) (MA-input-p sigs))))
  (b-if (b-and (DE-valid? (DQ-DE3 DQ)) (dispatch-inst? MA))
        (dispatch-entry 0 0 0 0 0 0 0 0 0)
  (b-if (bs-and (b-not (DE-valid? (DQ-DE3 DQ)))
                (DE-valid? (DQ-DE2 DQ))
                (b-not (dispatch-inst? MA)))
        (update-dispatch-entry (decode-output (MA-IFU MA) MA sigs)
           :valid? (b-andc2 (DE-valid? (decode-output (MA-IFU MA) MA sigs))
                            (flush-all? MA sigs)))
        (update-dispatch-entry (DQ-DE3 DQ)
                  :valid? (b-andc2 (DE-valid? (DQ-DE3 DQ))
                                   (flush-all? MA sigs))))))

; The next state of a register reference table entry.
(defun step-reg-ref (reg-ref idx MA sigs)
  (declare (xargs :guard (and (reg-ref-p reg-ref)
                              (rname-p idx)
                              (MA-state-p MA) (MA-input-p sigs))))
  (reg-ref (b-if (flush-all? MA sigs)
                 0
                 (b-if (bs-and (dispatch-inst? MA)
                               (cntlv-wb? (dispatch-cntlv MA))
                               (b-not (cntlv-wb-sreg? (dispatch-cntlv MA)))
                               (bv-eqv *rname-size*
                                       (dispatch-dest-reg MA)
                                       idx))
                       1
                       (b-if (bs-and (ROB-write-reg? (MA-ROB MA))
                                     (bv-eqv *rname-size*
                                             (ROB-write-rid (MA-ROB MA))
                                             idx)
                                     (bv-eqv *rob-index-size*
                                             (ROB-head (MA-ROB MA))
                                             (reg-ref-tag reg-ref)))
                             0 (reg-ref-wait? reg-ref))))
           (b-if (bs-and (dispatch-inst? MA)
                         (cntlv-wb? (dispatch-cntlv MA))
                         (b-not (cntlv-wb-sreg? (dispatch-cntlv MA)))
                         (bv-eqv *rname-size*
                                 (dispatch-dest-reg MA)
                                 idx))
                 (ROB-tail (MA-ROB MA))
                 (reg-ref-tag reg-ref))))

(defun step-reg-list (r-list idx MA sigs)
  (declare (xargs :guard (and (reg-ref-listp r-list)
                              (rname-p idx)
                              (MA-state-p MA)
                              (MA-input-p sigs))))
  (if (endp r-list)
      nil
      (cons (step-reg-ref (car r-list) idx MA sigs)
```

```
           (step-reg-list (cdr r-list) (rname (1+ idx)) MA sigs))))

; The next state of the register reference table.
(defun step-reg-tbl  (r-list MA sigs)
  (declare (xargs :guard (and (reg-tbl-p r-list)
                              (MA-state-p MA)
                              (MA-input-p sigs))
                  :guard-hints (("Goal" :in-theory (enable reg-tbl-p)))))
  (step-reg-list r-list 0 MA sigs))

(defthm reg-ref-listp-step-reg-list
    (implies (and (reg-ref-listp reg-tbl) (MA-state-p MA) (MA-input-p sigs))
             (reg-ref-listp (step-reg-list reg-tbl idx MA sigs))))

(defthm len-step-reg-list
    (equal (len (step-reg-list reg-tbl idx MA sigs))
           (len reg-tbl)))

(defthm reg-tbl-p-step-reg-tbl
    (implies (and (reg-tbl-p reg-tbl)
                  (MA-state-p MA) (MA-input-p sigs))
             (reg-tbl-p (step-reg-tbl reg-tbl MA sigs)))
  :hints (("goal" :in-theory (enable reg-tbl-p))))

(in-theory (disable step-reg-tbl))

; The next state of a special register reference table entry.
(defun step-sreg-ref (sreg-ref idx MA sigs)
  (declare (xargs :guard (and (reg-ref-p sreg-ref)
                              (rname-p idx)
                              (MA-state-p MA)
                              (MA-input-p sigs))))
  (reg-ref (b-if (flush-all? MA sigs) 0
                 (b-if (bs-and (dispatch-inst? MA)
                               (cntlv-wb? (dispatch-cntlv MA))
                               (cntlv-wb-sreg? (dispatch-cntlv MA))
                               (bv-eqv *rname-size*
                                       (dispatch-dest-reg MA)
                                       idx))
                       1
                       (b-if (bs-and (ROB-write-sreg? (MA-ROB MA))
                                     (bv-eqv *rname-size*
                                             (ROB-write-rid (MA-ROB MA))
                                             idx)
                                     (bv-eqv *rob-index-size*
                                             (ROB-head (MA-ROB MA))
                                             (reg-ref-tag sreg-ref)))
                             0
                             (reg-ref-wait? sreg-ref))))
           (b-if (bs-and (dispatch-inst? MA)
                         (cntlv-wb? (dispatch-cntlv MA))
                         (cntlv-wb-sreg? (dispatch-cntlv MA))
                         (bv-eqv *rname-size*
                                 (dispatch-dest-reg MA)
                                 idx))
                 (ROB-tail (MA-ROB MA))
                 (reg-ref-tag sreg-ref))))

; The next state of special register reference table.
(defun step-sreg-tbl (sreg-tbl MA sigs)
  (declare (xargs :guard (and (sreg-tbl-p sreg-tbl) (MA-state-p MA)
                              (MA-input-p sigs))))
```

```
  (sreg-tbl (step-sreg-ref (sreg-tbl-sr0 sreg-tbl) 0 MA sigs)
            (step-sreg-ref (sreg-tbl-sr1 sreg-tbl) 1 MA sigs)))

; The next state of DQ
(defun step-DQ (MA sigs)
  (declare (xargs :guard (and (MA-state-p MA) (MA-input-p sigs))))
  (DQ (step-DE0 (MA-DQ MA) MA sigs)
      (step-DE1 (MA-DQ MA) MA sigs)
      (step-DE2 (MA-DQ MA) MA sigs)
      (step-DE3 (MA-DQ MA) MA sigs)
      (step-reg-tbl (DQ-reg-tbl (MA-DQ MA)) MA sigs)
      (step-sreg-tbl (DQ-sreg-tbl (MA-DQ MA)) MA sigs)))

(defthm DQ-p-step-DQ
    (implies (and (MA-state-p MA) (MA-input-p sigs))
             (DQ-p (step-DQ MA sigs))))
(in-theory (disable step-DQ))

;Commit-inst? is on when the instruction at the head of the ROB commits.
; If the instruction at the head of the ROB is an exception causing
; instruction or synchronizing instruction, we check whether there are
; pending writes in the write buffer before allowing it to commit.
(defun commit-inst? (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (let ((ROB (MA-ROB MA))
        (LSU (MA-LSU MA)))
    (let ((robe (nth-robe (ROB-head ROB) ROB)))
      (bs-and (ROBE-valid? robe)
              (ROBE-complete? robe)
              (b-nand (b-ior (excpt-raised? (ROBE-excpt robe))
                             (ROBE-sync? robe))
                      (LSU-pending-writes? LSU))))))

(defthm bitp-commit-inst? (bitp (commit-inst? MA)))
(in-theory (disable commit-inst?))

; This value is 1 when the ROB entry designated by index should hold
; the instruction dispatched in the current cycle.
(defun ROBE-receive-inst? (ROB index MA)
  (declare (xargs :guard (and (ROB-p rob)
                              (rob-index-p index) (MA-state-p MA))))
  (b-and (dispatch-inst? MA)
         (bv-eqv *rob-index-size* (ROB-tail ROB) index)))

(defthm bitp-ROBE-receive-inst?
    (bitp (ROBE-receive-inst? ROB index MA)))
(in-theory (disable ROBE-receive-inst?))

; This value is 1 when the ROB entry designated by index should
; read the result of an instruction put on the CDB.
(defun ROBE-receive-result? (ROB index MA)
  (declare (xargs :guard (and (ROB-p rob) (rob-index-p index)
                              (MA-state-p MA))))
  (bs-and (ROBE-valid? (nth-ROBE index ROB))
          (CDB-ready-for? index MA)))

(defthm bitp-ROBE-receive-result?
    (bitp (ROBE-receive-result? ROB index MA)))
(in-theory (disable ROBE-receive-result?))

; The next state of an ROB entry designated by idx.
(defun step-ROBE (robe idx ROB MA sigs)
```

```
  (declare (xargs :guard (and (ROB-entry-p robe)
                              (rob-index-p idx)
                              (ROB-p ROB) (MA-state-p MA) (MA-input-p sigs))))
  (ROB-entry (b-if (b-ior (b-and (commit-inst? MA)
                                 (bv-eqv *rob-index-size* (ROB-head ROB) idx))
                          (flush-all? MA sigs))
                   0
                   (b-if (robe-receive-inst? ROB idx MA)
                         1
                         (ROBE-valid? robe))) ; valid?
             (b-if (robe-receive-inst? ROB idx MA)
                   (dispatch-no-unit? MA)
                   (b-ior (ROBE-complete? robe)
                          (robe-receive-result? rob idx MA))) ;complete?
             (b-if (robe-receive-inst? rob idx MA)
                   (dispatch-excpt MA)
                   (b-if (robe-receive-result? rob idx MA)
                         (CDB-excpt MA)
                         (ROBE-excpt robe))) ;excpt
             (b-if (robe-receive-inst? rob idx MA)
                   (cntlv-wb? (dispatch-cntlv MA))
                   (ROBE-wb? robe))
             (b-if (robe-receive-inst? ROB idx MA)
                   (cntlv-wb-sreg? (dispatch-cntlv MA))
                   (ROBE-wb-sreg? robe))
             (b-if (robe-receive-inst? ROB idx MA)
                   (cntlv-sync? (dispatch-cntlv MA))
                   (ROBE-sync? robe))
             (b-if (robe-receive-inst? ROB idx MA)
                   (logbit 3 (cntlv-exunit (dispatch-cntlv MA)))
                   (ROBE-branch? robe))
             (b-if (robe-receive-inst? ROB idx MA)
                   (cntlv-rfeh? (dispatch-cntlv MA))
                   (ROBE-rfeh? robe))
             (b-if (robe-receive-inst? ROB idx MA)
                   (cntlv-br-predict? (dispatch-cntlv MA))
                   (ROBE-br-predict? robe))
             (b-if (b-and (robe-receive-result? rob idx MA)
                          (ROBE-branch? robe))
                   (logcar (CDB-val MA))
                   (ROBE-br-actual? robe))
             (b-if (robe-receive-inst? ROB idx MA)
                   (dispatch-pc MA)
                   (ROBE-pc robe))
             (b-if (robe-receive-inst? ROB idx MA)
                   (word (DE-br-target (DQ-DE0 (MA-DQ MA))))
                   (b-if (b-andc2 (robe-receive-result? rob idx MA)
                                  (ROBE-branch? robe))
                         (CDB-val MA)
                         (ROBE-val robe)))
             (b-if (robe-receive-inst? ROB idx MA)
                   (dispatch-dest-reg MA)
                   (ROBE-dest robe))))

(defthm ROBE-p-step-robe
    (implies (and (ROB-entry-p robe)
                  (ROB-p ROB)
                  (MA-state-p MA)
                  (MA-input-p sigs))
             (ROB-entry-p (step-robe robe index ROB MA sigs))))
(in-theory (disable step-robe))
```

```
(defun step-ROBE-list (entries index ROB MA sigs)
  (declare (xargs :guard (and (ROBE-listp entries)
                              (rob-index-p index)
                              (ROB-p ROB) (MA-state-p MA) (MA-input-p sigs))))
  (if (endp entries)
      nil
      (cons (step-ROBE (car entries) index ROB MA sigs)
            (step-ROBE-list (cdr entries) (rob-index (1+ index))
                            ROB MA sigs))))

(defun step-ROB-entries (entries ROB MA sigs)
  (declare (xargs :guard (and (ROB-entries-p entries)
                              (ROB-p ROB) (MA-state-p MA) (MA-input-p sigs))
                  :guard-hints (("Goal" :in-theory (enable ROB-entries-p)))))
  (step-ROBE-list entries 0 ROB MA sigs))

(defthm robe-lisp-step-robe-list
    (implies (and (robe-listp entries)
                  (rob-p rob)
                  (MA-state-p MA)
                  (MA-input-p sigs))
             (ROBE-listp (step-robe-list entries index rob MA sigs))))

(defthm len-step-ROBE-list
    (equal (len (step-ROBE-list entries index ROB MA sigs))
           (len entries)))

(defthm ROB-entries-p-step-ROB-entries
    (implies (and (ROB-entries-p entries)
                  (ROB-p ROB)
                  (MA-state-p MA)
                  (MA-input-p sigs))
             (ROB-entries-p (step-ROB-entries entries ROB MA sigs)))
  :hints (("Goal" :in-theory (enable rob-entries-p))))
(in-theory (disable step-ROB-entries))

; The next state of the ROB.
(defun step-ROB (MA sigs)
  (declare (xargs :guard (and (MA-state-p MA) (MA-input-p sigs))))
  (ROB (b-if (flush-all? MA sigs) 0
             (b-xor (ROB-flg (MA-ROB MA))
                    (b-xor (b-and (commit-inst? MA)
                                  (logbit *rob-index-size*
                                          (+ 1 (ROB-head (MA-ROB MA)))))
                           (b-and (dispatch-inst? MA)
                                  (logbit *rob-index-size*
                                          (+ 1 (ROB-tail (MA-ROB MA)))))))))
       (b-if (b-ior (enter-excpt? MA) (ex-intr? MA sigs)) 0
             (b-ior (MA-input-exintr sigs)
                    (ROB-exintr? (MA-ROB MA))))
       (b-if (flush-all? MA sigs) 0
             (b-if (commit-inst? MA)
                   (rob-index (+ 1 (ROB-head (MA-ROB MA))))
                   (ROB-head (MA-ROB MA))))
       (b-if (flush-all? MA sigs) 0
             (b-if (dispatch-inst? MA)
                   (rob-index (+ 1 (ROB-tail (MA-ROB MA))))
                   (ROB-tail (MA-ROB MA))))
       (step-ROB-entries (ROB-entries (MA-ROB MA)) (MA-ROB MA) MA sigs)))

(defthm ROB-p-step-ROB
    (implies (and (MA-state-p MA) (MA-input-p sigs))
```

```
            (ROB-p (step-ROB MA sigs))))
(in-theory (disable step-ROB))

; The next state of IU-RS0.
(defun step-IU-RS0 (iu MA sigs)
  (declare (xargs :guard (and (integer-unit-p IU)
                              (MA-state-p MA) (MA-input-p sigs))))
  (let ((RS0 (IU-RS0 iu)))
    (RS (b-andc1 (flush-all? MA sigs)
                 (b-if (RS-valid? rs0)
                       (b-not (issue-IU-RS0? iu MA))
                       (b-and (dispatch-to-IU? MA) (select-IU-RS0? iu))))
        (b-if (b-and (dispatch-to-IU? MA) (select-IU-RS0? iu))
              (b-not (logbit 0 (cntlv-operand (dispatch-cntlv MA))))
              (RS-op RS0))
        (b-if (b-and (dispatch-to-IU? MA) (select-IU-RS0? iu))
              (ROB-tail (MA-ROB MA))
              (RS-tag RS0))
        (b-if (b-and (dispatch-to-IU? MA) (select-IU-RS0? iu))
              (dispatch-ready1? MA)
              (b-ior (RS-ready1? RS0)
                     (CDB-ready-for? (RS-src1 RS0) MA)))
        (b-if (b-and (dispatch-to-IU? MA) (select-IU-RS0? iu))
              (dispatch-ready2? MA)
              (b-ior (RS-ready2? RS0)
                     (CDB-ready-for? (RS-src2 RS0) MA)))
        (b-if (b-and (dispatch-to-IU? MA) (select-IU-RS0? iu))
              (dispatch-val1 MA)
              (b-if (b-andc1 (RS-ready1? RS0)
                             (CDB-ready-for? (RS-src1 RS0) MA))
                    (CDB-val MA)
                    (RS-val1 RS0)))
        (b-if (b-and (dispatch-to-IU? MA) (select-IU-RS0? iu))
              (dispatch-val2 MA)
              (b-if (b-andc1 (RS-ready2? RS0)
                             (CDB-ready-for? (RS-src2 RS0) MA))
                    (CDB-val MA)
                    (RS-val2 RS0)))
        (b-if (b-and (dispatch-to-IU? MA) (select-IU-RS0? iu))
              (dispatch-src1 MA)
              (RS-src1 RS0))
        (b-if (b-and (dispatch-to-IU? MA) (select-IU-RS0? iu))
              (dispatch-src2 MA)
              (RS-src2 RS0)))))

(defthm RS-p-step-IU-RS0
    (implies (and (integer-unit-p IU) (MA-state-p MA) (MA-input-p sigs))
             (RS-p (step-IU-RS0 IU MA sigs))))
(in-theory (disable step-IU-RS0))

; The next state of IU-RS1.
(defun step-IU-RS1 (iu MA sigs)
  (declare (xargs :guard (and (integer-unit-p IU)
                              (MA-state-p MA) (MA-input-p sigs))))
  (let ((RS1 (IU-RS1 iu)))
    (RS (b-andc1 (flush-all? MA sigs)
                 (b-if (RS-valid? rs1)
                       (b-not (issue-IU-RS1? iu MA))
                       (b-and (dispatch-to-IU? MA) (select-IU-RS1? iu))))
        (b-if (b-and (dispatch-to-IU? MA) (select-IU-RS1? iu))
              (b-not (logbit 0 (cntlv-operand (dispatch-cntlv MA))))
              (RS-op RS1))
```

```
        (b-if (b-and (dispatch-to-IU? MA) (select-IU-RS1? iu))
              (ROB-tail (MA-ROB MA))
              (RS-tag RS1))
        (b-if (b-and (dispatch-to-IU? MA) (select-IU-RS1? iu))
              (dispatch-ready1? MA)
              (b-ior (RS-ready1? RS1)
                     (CDB-ready-for? (RS-src1 RS1) MA)))
        (b-if (b-and (dispatch-to-IU? MA) (select-IU-RS1? iu))
              (dispatch-ready2? MA)
              (b-ior (RS-ready2? RS1)
                     (CDB-ready-for? (RS-src2 RS1) MA)))
        (b-if (b-and (dispatch-to-IU? MA) (select-IU-RS1? iu))
              (dispatch-val1 MA)
              (b-if (b-andc1 (RS-ready1? RS1)
                             (CDB-ready-for? (RS-src1 RS1) MA))
                    (CDB-val MA)
                    (RS-val1 RS1)))
        (b-if (b-and (dispatch-to-IU? MA) (select-IU-RS1? iu))
              (dispatch-val2 MA)
              (b-if (b-andc1 (RS-ready2? RS1)
                             (CDB-ready-for? (RS-src2 RS1) MA))
                    (CDB-val MA)
                    (RS-val2 RS1)))
        (b-if (b-and (dispatch-to-IU? MA) (select-IU-RS1? iu))
              (dispatch-src1 MA)
              (RS-src1 RS1))
        (b-if (b-and (dispatch-to-IU? MA) (select-IU-RS1? iu))
              (dispatch-src2 MA)
              (RS-src2 RS1)))))

(defthm RS-p-step-IU-RS1
    (implies (and (integer-unit-p IU) (MA-state-p MA) (MA-input-p sigs))
             (RS-p (step-IU-RS1 IU MA sigs))))
(in-theory (disable step-IU-RS1))

; The next state of IU.
(defun step-IU (MA sigs)
  (declare (xargs :guard (and (MA-state-p MA) (MA-input-p sigs))))
  (integer-unit (step-IU-RS0 (MA-IU MA) MA sigs)
                (step-IU-RS1 (MA-IU MA) MA sigs)))

(defthm integer-unit-p-step-IU
    (implies (and (MA-state-p MA) (MA-input-p sigs))
             (integer-unit-p (step-IU MA sigs))))
(in-theory (disable step-IU))

; The next state of reservation station MU-RS0.
(defun step-MU-RS0 (MU MA sigs)
  (declare (xargs :guard (and (mult-unit-p MU)
                              (MA-state-p MA) (MA-input-p sigs))))
  (let ((RS0 (MU-RS0 MU)))
    (RS (b-andc1 (flush-all? MA sigs)
                 (b-if (RS-valid? rs0)
                       (b-not (issue-MU-RS0? MU MA))
                       (b-and (dispatch-to-MU? MA)
                              (select-MU-RS0? MU))))
        0
        (b-if (b-and (dispatch-to-MU? MA) (select-MU-RS0? MU))
              (ROB-tail (MA-ROB MA))
              (RS-tag RS0))
        (b-if (b-and (dispatch-to-MU? MA) (select-MU-RS0? MU))
              (dispatch-ready1? MA)
```

```
        (b-ior (RS-ready1? RS0)
               (CDB-ready-for? (RS-src1 RS0) MA)))
      (b-if (b-and (dispatch-to-MU? MA) (select-MU-RS0? MU))
        (dispatch-ready2? MA)
        (b-ior (RS-ready2? RS0)
               (CDB-ready-for? (RS-src2 RS0) MA)))
      (b-if (b-and (dispatch-to-MU? MA) (select-MU-RS0? MU))
        (dispatch-val1 MA)
        (b-if (b-andc1 (RS-ready1? RS0)
                       (CDB-ready-for? (RS-src1 RS0) MA))
              (CDB-val MA)
              (RS-val1 RS0)))
      (b-if (b-and (dispatch-to-MU? MA) (select-MU-RS0? MU))
        (dispatch-val2 MA)
        (b-if (b-andc1 (RS-ready2? RS0)
                       (CDB-ready-for? (RS-src2 RS0) MA))
              (CDB-val MA)
              (RS-val2 RS0)))
      (b-if (b-and (dispatch-to-MU? MA) (select-MU-RS0? MU))
        (dispatch-src1 MA)
        (RS-src1 RS0))
      (b-if (b-and (dispatch-to-MU? MA) (select-MU-RS0? MU))
        (dispatch-src2 MA)
        (RS-src2 RS0)))))

; The next state of reservation station MU-RS1.
(defun step-MU-RS1 (MU MA sigs)
  (declare (xargs :guard (and (mult-unit-p MU)
                              (MA-state-p MA) (MA-input-p sigs))))
  (let ((RS1 (MU-RS1 MU)))
    (RS (b-andc1 (flush-all? MA sigs)
                 (b-if (RS-valid? rs1)
                       (b-not (issue-MU-RS1? MU MA))
                       (b-and (dispatch-to-MU? MA)
                              (select-MU-RS1? MU))))
        0
        (b-if (b-and (dispatch-to-MU? MA) (select-MU-RS1? MU))
          (ROB-tail (MA-ROB MA))
          (RS-tag RS1))
        (b-if (b-and (dispatch-to-MU? MA) (select-MU-RS1? MU))
          (dispatch-ready1? MA)
          (b-ior (RS-ready1? RS1)
                 (CDB-ready-for? (RS-src1 RS1) MA)))
        (b-if (b-and (dispatch-to-MU? MA) (select-MU-RS1? MU))
          (dispatch-ready2? MA)
          (b-ior (RS-ready2? RS1)
                 (CDB-ready-for? (RS-src2 RS1) MA)))
        (b-if (b-and (dispatch-to-MU? MA) (select-MU-RS1? MU))
          (dispatch-val1 MA)
          (b-if (b-andc1 (RS-ready1? RS1)
                         (CDB-ready-for? (RS-src1 RS1) MA))
                (CDB-val MA)
                (RS-val1 RS1)))
        (b-if (b-and (dispatch-to-MU? MA) (select-MU-RS1? MU))
          (dispatch-val2 MA)
          (b-if (b-andc1 (RS-ready2? RS1)
                         (CDB-ready-for? (RS-src2 RS1) MA))
                (CDB-val MA)
                (RS-val2 RS1)))
        (b-if (b-and (dispatch-to-MU? MA) (select-MU-RS1? MU))
          (dispatch-src1 MA)
          (RS-src1 RS1))
```

```
        (b-if (b-and (dispatch-to-MU? MA) (select-MU-RS1? MU))
              (dispatch-src2 MA)
              (RS-src2 RS1)))))

; The next state of latch MU-lch1.
(defun step-MU-lch1 (MU MA sigs)
  (declare (xargs :guard (and (mult-unit-p MU)
                              (MA-state-p MA) (MA-input-p sigs))))
  (b-if (bs-ior (CDB-for-MU? MA)
                (b-not (MU-latch1-valid? (MU-lch1 MU)))
                (b-not (MU-latch2-valid? (MU-lch2 MU))))
        (b-if (issue-MU-RS0? MU MA)
              (MU-latch1 (b-not (flush-all? MA sigs))
                         (RS-tag (MU-RS0 MU))
                         (ML1-output (RS-val1 (MU-RS0 MU))
                                     (RS-val2 (MU-RS0 MU))))
              (b-if (issue-MU-RS1? MU MA)
                    (MU-latch1 (b-not (flush-all? MA sigs))
                               (RS-tag (MU-RS1 MU))
                               (ML1-output (RS-val1 (MU-RS1 MU))
                                           (RS-val2 (MU-RS1 MU))))
                    (MU-latch1 0 0 (ML1-output (RS-val1 (MU-RS1 MU))
                                               (RS-val2 (MU-RS1 MU))))))
        (update-MU-latch1 (MU-lch1 MU)
                          :valid? (b-andc1 (flush-all? MA sigs)
                                           (MU-latch1-valid? (MU-lch1 MU))))))

; The next state of latch MU-lch2.
(defun step-MU-lch2 (MU MA sigs)
  (declare (xargs :guard (and (mult-unit-p MU)
                              (MA-state-p MA) (MA-input-p sigs))))
  (b-if (b-orc2 (CDB-for-MU? MA)
                (MU-latch2-valid? (MU-lch2 MU)))
        (MU-latch2 (b-and (MU-latch1-valid? (MU-lch1 MU))
                          (b-not (flush-all? MA sigs)))
                   (MU-latch1-tag (MU-lch1 MU))
                   (ML2-output (MU-latch1-data (MU-lch1 MU))))
        (update-MU-latch2 (MU-lch2 MU)
                          :valid? (b-andc1 (flush-all? MA sigs)
                                           (MU-latch2-valid? (MU-lch2 MU))))))

; The next state of the MU.
(defun step-MU (MA sigs)
  (declare (xargs :guard (and (MA-state-p MA) (MA-input-p sigs))))
  (mult-unit (step-MU-RS0 (MA-MU MA) MA sigs)
             (step-MU-RS1 (MA-MU MA) MA sigs)
             (step-MU-lch1 (MA-MU MA) MA sigs)
             (step-MU-lch2 (MA-MU MA) MA sigs)))

(defthm mult-unit-p-step-MU
    (implies (and (MA-state-p MA) (MA-input-p sigs))
             (mult-unit-p (step-MU MA sigs))))
(in-theory (disable step-MU))

; The next state of reservation station BU-RS0.
(defun step-BU-RS0 (BU MA sigs)
  (declare (xargs :guard (and (branch-unit-p BU)
                              (MA-state-p MA) (MA-input-p sigs))))
  (let ((RS0 (BU-RS0 BU)))
    (BU-RS (b-and (b-not (flush-all? MA sigs))
                  (b-if (BU-RS-valid? RS0)
                        (b-not (issue-BU-RS0? BU MA))
```

```
                    (b-and (dispatch-to-BU? MA) (select-BU-RS0? BU))))
            (b-if (b-and (dispatch-to-BU? MA) (select-BU-RS0? BU))
                  (ROB-tail (MA-ROB MA))
                  (BU-RS-tag RS0))
            (b-if (b-and (dispatch-to-BU? MA) (select-BU-RS0? BU))
                  (dispatch-ready3? MA)
                  (b-ior (BU-RS-ready? RS0)
                         (CDB-ready-for? (BU-RS-src RS0) MA)))
            (b-if (b-and (dispatch-to-BU? MA) (select-BU-RS0? BU))
                  (dispatch-val3 MA)
                  (b-if (b-andc1 (BU-RS-ready? RS0)
                                 (CDB-ready-for? (BU-RS-src RS0) MA))
                        (CDB-val MA)
                        (BU-RS-val RS0)))
            (b-if (b-and (dispatch-to-BU? MA) (select-BU-RS0? BU))
                  (dispatch-src3 MA)
                  (BU-RS-src RS0)))))

; The next state of reservation station BU-RS1.
(defun step-BU-RS1 (BU MA sigs)
  (declare (xargs :guard (and (branch-unit-p BU)
                              (MA-state-p MA) (MA-input-p sigs))))
  (let ((RS1 (BU-RS1 BU)))
    (BU-RS (b-and (b-not (flush-all? MA sigs))
                  (b-if (BU-RS-valid? RS1)
                        (b-not (issue-BU-RS1? BU MA))
                        (b-and (dispatch-to-BU? MA) (select-BU-RS1? BU))))
           (b-if (b-and (dispatch-to-BU? MA) (select-BU-RS1? BU))
                 (ROB-tail (MA-ROB MA))
                 (BU-RS-tag RS1))
           (b-if (b-and (dispatch-to-BU? MA) (select-BU-RS1? BU))
                 (dispatch-ready3? MA)
                 (b-ior (BU-RS-ready? RS1)
                        (CDB-ready-for? (BU-RS-src RS1) MA)))
           (b-if (b-and (dispatch-to-BU? MA) (select-BU-RS1? BU))
                 (dispatch-val3 MA)
                 (b-if (b-andc1 (BU-RS-ready? RS1)
                                (CDB-ready-for? (BU-RS-src RS1) MA))
                       (CDB-val MA)
                       (BU-RS-val RS1)))
           (b-if (b-and (dispatch-to-BU? MA) (select-BU-RS1? BU))
                 (dispatch-src3 MA)
                 (BU-RS-src RS1)))))

; The next state of the BU.
(defun step-BU (MA sigs)
  (declare (xargs :guard (and (MA-state-p MA) (MA-input-p sigs))))
  (branch-unit (step-BU-RS0 (MA-BU MA) MA sigs)
               (step-BU-RS1 (MA-BU MA) MA sigs)))

(defthm branch-unit-p-step-BU
    (implies (and (MA-state-p MA) (MA-input-p sigs))
             (branch-unit-p (step-BU MA sigs))))
(in-theory (disable step-BU))

; The next state of the rs1-head?, which is a flag in the LSU to determine
; which reservation station contains the first load-store instruction.
; The order of load-store instruction is important, as the Tomasulo's
; algorithm does not guarantee the correctness of the execution.
(defun step-rs1-head? (LSU MA sigs)
  (declare (xargs :guard (and (load-store-unit-p LSU)
                              (MA-state-p MA) (MA-input-p sigs))))
```

```
  (b-if (issue-LSU-RS0? LSU MA sigs) 1
  (b-if (issue-LSU-RS1? LSU MA sigs) 0
        (LSU-rs1-head? LSU))))

(defthm bitp-step-rs1-head?
    (implies (and (load-store-unit-p LSU)
                  (MA-state-p MA) (MA-input-p sigs))
             (bitp (step-rs1-head? LSU MA sigs))))
(in-theory (disable step-rs1-head?))

; The next state of reservation station LSU-RS0.  Remember reservation
; stations for the LSU forms a queue as our machine needs to know the
; exact order of memory accesses.  RS0 is the head of the queue.
(defun step-LSU-RS0 (LSU MA sigs)
  (declare (xargs :guard (and (load-store-unit-p LSU)
                              (MA-state-p MA) (MA-input-p sigs))))
  (let ((RS0 (LSU-RS0 LSU)))
    (LSU-RS (b-and (b-not (flush-all? MA sigs))
                   (b-if (LSU-RS-valid? RS0)
                         (b-not (issue-LSU-RS0? LSU MA sigs))
                         (b-and (dispatch-to-LSU? MA)
                                (select-LSU-RS0? LSU))))
            (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS0? LSU))
                  (logbit 1 (cntlv-operand (dispatch-cntlv MA)))
                  (LSU-RS-op RS0))
            (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS0? LSU))
                  (cntlv-ld-st? (dispatch-cntlv MA))
                  (LSU-RS-ld-st? RS0))
            (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS0? LSU))
                  (ROB-tail (MA-ROB MA))
                  (LSU-RS-tag RS0))
            (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS0? LSU))
                  (dispatch-ready3? MA)
                  (b-ior (LSU-RS-rdy3? RS0)
                         (CDB-ready-for? (LSU-RS-src3 RS0) MA)))
            (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS0? LSU))
                  (dispatch-val3 MA)
                  (b-if (b-andc1 (LSU-RS-rdy3? RS0)
                                 (CDB-ready-for? (LSU-RS-src3 RS0) MA))
                        (CDB-val MA)
                        (LSU-RS-val3 RS0)))
            (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS0? LSU))
                  (dispatch-src3 MA)
                  (LSU-RS-src3 RS0))
            (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS0? LSU))
                  (dispatch-ready1? MA)
                  (b-ior (LSU-RS-rdy1? RS0)
                         (CDB-ready-for? (LSU-RS-src1 RS0) MA)))
            (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS0? LSU))
                  (dispatch-val1 MA)
                  (b-if (b-andc1 (LSU-RS-rdy1? RS0)
                                 (CDB-ready-for? (LSU-RS-src1 RS0) MA))
                        (CDB-val MA)
                        (LSU-RS-val1 RS0)))
            (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS0? LSU))
                  (dispatch-src1 MA)
                  (LSU-RS-src1 RS0))
            (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS0? LSU))
                  (dispatch-ready2? MA)
                  (bs-ior (LSU-RS-rdy2? RS0)
                          (CDB-ready-for? (LSU-RS-src2 RS0) MA)))
            (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS0? LSU))
```

```
                (dispatch-val2 MA)
                (b-if (b-andc1 (LSU-RS-rdy2? RS0)
                               (CDB-ready-for? (LSU-RS-src2 RS0) MA))
                      (CDB-val MA)
                      (LSU-RS-val2 RS0)))
          (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS0? LSU))
                (dispatch-src2 MA)
                (LSU-RS-src2 RS0)))))

(defthm LSU-RS-p-step-LSU-RS0
    (implies (and (load-store-unit-p LSU)
                  (MA-state-p MA)
                  (MA-input-p sigs))
             (LSU-RS-p (step-LSU-RS0 LSU MA sigs))))
(in-theory (disable step-LSU-RS0))

; The next state of reservation station LSU-RS1.
(defun step-LSU-RS1 (LSU MA sigs)
  (declare (xargs :guard (and (load-store-unit-p LSU)
                              (MA-state-p MA) (MA-input-p sigs))))
  (let ((RS1 (LSU-RS1 LSU)))
    (LSU-RS (b-and (b-not (flush-all? MA sigs))
                   (b-if (LSU-RS-valid? RS1)
                         (b-not (issue-LSU-RS1? LSU MA sigs))
                         (b-and (dispatch-to-LSU? MA)
                                (select-LSU-RS1? LSU))))
            (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS1? LSU))
                  (logbit 1 (cntlv-operand (dispatch-cntlv MA)))
                  (LSU-RS-op RS1))
            (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS1? LSU))
                  (cntlv-ld-st? (dispatch-cntlv MA))
                  (LSU-RS-ld-st? RS1))
            (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS1? LSU))
                  (ROB-tail (MA-ROB MA))
                  (LSU-RS-tag RS1))
            (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS1? LSU))
                  (dispatch-ready3? MA)
                  (b-ior (LSU-RS-rdy3? RS1)
                         (CDB-ready-for? (LSU-RS-src3 RS1) MA)))
            (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS1? LSU))
                  (dispatch-val3 MA)
                  (b-if (b-andc1 (LSU-RS-rdy3? RS1)
                                 (CDB-ready-for? (LSU-RS-src3 RS1) MA))
                        (CDB-val MA)
                        (LSU-RS-val3 RS1)))
            (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS1? LSU))
                  (dispatch-src3 MA)
                  (LSU-RS-src3 RS1))
            (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS1? LSU))
                  (dispatch-ready1? MA)
                  (b-ior (LSU-RS-rdy1? RS1)
                         (CDB-ready-for? (LSU-RS-src1 RS1) MA)))
            (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS1? LSU))
                  (dispatch-val1 MA)
                  (b-if (b-andc1 (LSU-RS-rdy1? RS1)
                                 (CDB-ready-for? (LSU-RS-src1 RS1) MA))
                        (CDB-val MA)
                        (LSU-RS-val1 RS1)))
            (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS1? LSU))
                  (dispatch-src1 MA)
                  (LSU-RS-src1 RS1))
            (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS1? LSU))
```

```
                (dispatch-ready2? MA)
                (b-ior (LSU-RS-rdy2? RS1)
                       (CDB-ready-for? (LSU-RS-src2 RS1) MA)))
          (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS1? LSU))
                (dispatch-val2 MA)
                (b-if (b-andc1 (LSU-RS-rdy2? RS1)
                               (CDB-ready-for? (LSU-RS-src2 RS1) MA))
                      (CDB-val MA)
                      (LSU-RS-val2 RS1)))
          (b-if (b-and (dispatch-to-LSU? MA) (select-LSU-RS1? LSU))
                (dispatch-src2 MA)
                (LSU-RS-src2 RS1)))))

(defthm LSU-RS-p-step-LSU-RS1
    (implies (and (load-store-unit-p LSU)
                  (MA-state-p MA)
                  (MA-input-p sigs))
             (LSU-RS-p (step-LSU-RS1 LSU MA sigs))))
(in-theory (disable step-LSU-RS1))

; The next state of read buffer.
(defun step-rbuf (LSU MA sigs)
  (declare (xargs :guard (and (load-store-unit-p LSU)
                              (MA-state-p MA) (MA-input-p sigs))))
  (let ((RS0 (LSU-RS0 LSU)) (RS1 (LSU-RS1 LSU)))
    (b-if (b-andc2 (rbuf-valid? (LSU-rbuf LSU))
                   (release-rbuf? LSU MA sigs))
          (update-read-buffer (LSU-rbuf LSU)
                              :valid? (b-not (flush-all? MA sigs)))
          (b-if (b-andc2 (issue-LSU-RS0? LSU MA sigs)
                         (LSU-RS-ld-st? RS0))
                (read-buffer (b-not (flush-all? MA sigs))
                             (LSU-RS-tag RS0)
                             (b-if (LSU-RS-op RS0)
                                   (addr (LSU-RS-val1 RS0))
                                   (addr (+ (LSU-RS-val1 RS0)
                                            (LSU-RS-val2 RS0))))
                             (b-if (release-wbuf0? LSU sigs)
                                   (wbuf-valid? (LSU-wbuf1 LSU))
                                   (wbuf-valid? (LSU-wbuf0 LSU)))
                             (b-if (release-wbuf0? LSU sigs)
                                   0
                                   (wbuf-valid? (LSU-wbuf1 LSU))))
                (b-if (b-andc2 (issue-LSU-RS1? LSU MA sigs)
                               (LSU-RS-ld-st? RS1))
                      (read-buffer (b-not (flush-all? MA sigs))
                                   (LSU-RS-tag RS1)
                                   (b-if (LSU-RS-op RS1)
                                         (addr (LSU-RS-val1 RS1))
                                         (addr (+ (LSU-RS-val1 RS1)
                                                  (LSU-RS-val2 RS1))))
                                   (b-if (release-wbuf0? LSU sigs)
                                         (wbuf-valid? (LSU-wbuf1 LSU))
                                         (wbuf-valid? (LSU-wbuf0 LSU)))
                                   (b-if (release-wbuf0? LSU sigs)
                                         0
                                         (wbuf-valid? (LSU-wbuf1 LSU))))
                      (read-buffer 0 0 0 0 0))))))

(defthm read-buffer-p-step-rbuf
    (implies (and (load-store-unit-p LSU)
                  (MA-state-p MA)
```

```
                (MA-input-p sigs))
           (read-buffer-p (step-rbuf LSU MA sigs))))
(in-theory (disable step-rbuf))

; The next state of the result latch of the LSU.
(defun step-LSU-lch (LSU MA sigs)
  (declare (xargs :guard (and (load-store-unit-p LSU)
                              (MA-state-p MA) (MA-input-p sigs))))
  (LSU-latch (b-and (b-not (flush-all? MA sigs))
                    (bs-ior (release-rbuf? LSU MA sigs)
                            (check-wbuf0? LSU)
                            (check-wbuf1? LSU)))
             (b-if (b-ior (b-and (release-rbuf? LSU MA sigs)
                                 (LSU-read-prohibited? LSU (MA-mem MA)
                                                       (SRF-su (MA-SRF MA))))
                          (b-and (b-ior (check-wbuf0? LSU)
                                        (check-wbuf1? LSU))
                                 (LSU-write-prohibited? LSU (MA-mem MA)
                                                        (SRF-su (MA-SRF MA)))))
                   #b110 0)
             (b-if (release-rbuf? LSU MA sigs)
                   (rbuf-tag (LSU-rbuf LSU))
                   (b-if (check-wbuf0? LSU) (wbuf-tag (LSU-wbuf0 LSU))
                         (b-if (check-wbuf1? LSU) (wbuf-tag (LSU-wbuf1 LSU))
                               0)))
             (b-if (b-and (rbuf-valid? (LSU-rbuf LSU))
                          (LSU-address-match? LSU))
                   (LSU-forward-wbuf LSU)
                   (b-if (bs-and (rbuf-valid? (LSU-rbuf LSU))
                                 (b-not (LSU-read-prohibited?
                                         LSU (MA-mem MA)
                                         (SRF-su (MA-SRF MA))))
                                 (b-not (LSU-read-stall? LSU sigs)))
                         (read-mem (rbuf-addr (LSU-rbuf LSU)) (MA-mem MA))
                         0))))

(defthm LSU-latch-p-step-LSU-lch
    (implies (and (load-store-unit-p LSU)
                  (MA-state-p MA)
                  (MA-input-p sigs))
             (LSU-latch-p (step-LSU-lch LSU MA sigs))))
(in-theory (disable step-LSU-lch))

; The new write buffer entry value.  This represents the instruction
; issued from one of the reservation stations attached to the LSU.
(defun issued-write (LSU MA sigs)
  (declare (xargs :guard (and (load-store-unit-p LSU)
                              (MA-state-p MA) (MA-input-p sigs))))
  (let ((RS0 (LSU-RS0 LSU)) (RS1 (LSU-RS1 LSU)))
    (b-if (issue-LSU-RS0? LSU MA sigs)
          (write-buffer (b-andc2 (LSU-RS-ld-st? RS0)
                                 (flush-all? MA sigs))
                        0 0 (LSU-RS-tag RS0)
                        (b-if (LSU-RS-op RS0)
                              (addr (LSU-RS-val1 RS0))
                              (addr (+ (LSU-RS-val1 RS0) (LSU-RS-val2 RS0))))
                        (LSU-RS-val3 RS0))
    (b-if (issue-LSU-RS1? LSU MA sigs)
          (write-buffer (b-andc2 (LSU-RS-ld-st? RS1)
                                 (flush-all? MA sigs))
                        0 0 (LSU-RS-tag RS1)
                        (b-if (LSU-RS-op RS1)
```

```
                              (addr (LSU-RS-val1 RS1))
                              (addr (+ (LSU-RS-val1 RS1) (LSU-RS-val2 RS1))))
                        (LSU-RS-val3 RS1))
          (write-buffer 0 0 0 0 0 0)))))

(defthm write-buffer-p-issued-write
    (implies (and (load-store-unit-p LSU)
                  (MA-state-p MA)
                  (MA-input-p sigs))
             (write-buffer-p (issued-write LSU MA sigs))))
(in-theory (disable issued-write))

; Write buffer is designed in the same way as DQ is designed.
; The content of the wbuf1 is passed to the write buffer 0.  Wbuf1-output
; represents the trickle-down value from wbuf1 to wbuf0.
(defun wbuf1-output (LSU MA sigs)
  (declare (xargs :guard (and (load-store-unit-p LSU)
                              (MA-state-p MA) (MA-input-p sigs))))
  (let ((wbuf1 (LSU-wbuf1 LSU)))
    (b-if (wbuf-valid? wbuf1)
          (write-buffer
           (b-andc2 (wbuf-valid? wbuf1)
                    (b-andc2 (flush-all? MA sigs) (wbuf-commit? wbuf1)))
           (wbuf-complete? wbuf1)
           (b-ior (wbuf-commit? wbuf1)
                  (b-and (commit-inst? MA)
                         (bv-eqv *rob-index-size*
                                 (ROB-head (MA-ROB MA)) (wbuf-tag wbuf1))))
           (wbuf-tag wbuf1)
           (wbuf-addr wbuf1)
           (wbuf-val wbuf1))
          (issued-write LSU MA sigs))))

(defthm write-buffer-p-wbuf1-output
    (implies (and (load-store-unit-p LSU)
                  (MA-state-p MA)
                  (MA-input-p sigs))
             (write-buffer-p (wbuf1-output LSU MA sigs))))
(in-theory (disable write-buffer-p))

; The updated wbuf0 entry when the stored write is not released.
(defun update-wbuf0 (LSU MA sigs)
  (declare (xargs :guard (and (load-store-unit-p LSU)
                              (MA-state-p MA) (MA-input-p sigs))))
  (let ((wbuf0 (LSU-wbuf0 LSU)))
    (write-buffer (b-andc2 (wbuf-valid? wbuf0)
                           (b-andc2 (flush-all? MA sigs) (wbuf-commit? wbuf0)))
                  (b-ior (wbuf-complete? wbuf0) (check-wbuf0? LSU))
                  (b-ior (wbuf-commit? wbuf0)
                         (b-and (commit-inst? MA)
                                (bv-eqv *rob-index-size*
                                        (ROB-head (MA-ROB MA))
                                        (wbuf-tag wbuf0))))
                  (wbuf-tag wbuf0)
                  (wbuf-addr wbuf0)
                  (wbuf-val wbuf0))))

(defthm write-buffer-p-update-wbuf0
    (implies (and (load-store-unit-p LSU)
                  (MA-state-p MA)
                  (MA-input-p sigs))
             (write-buffer-p (update-wbuf0 LSU MA sigs))))
```

```
(in-theory (disable update-wbuf0))

; The updated wbuf1 entry when the stored write instruction is not
; passed to wbuf0.
(defun update-wbuf1 (LSU MA sigs)
  (declare (xargs :guard (and (load-store-unit-p LSU)
                              (MA-state-p MA) (MA-input-p sigs))))
  (let ((wbuf1 (LSU-wbuf1 LSU)))
    (write-buffer (b-andc2 (wbuf-valid? wbuf1)
                           (b-andc2 (flush-all? MA sigs) (wbuf-commit? wbuf1)))
                  (b-ior (wbuf-complete? wbuf1) (check-wbuf1? LSU))
                  (b-ior (wbuf-commit? wbuf1)
                         (b-and (commit-inst? MA)
                                (bv-eqv *rob-index-size*
                                        (ROB-head (MA-ROB MA))
                                        (wbuf-tag wbuf1))))
                  (wbuf-tag wbuf1)
                  (wbuf-addr wbuf1)
                  (wbuf-val wbuf1))))

(defthm write-buffer-p-update-wbuf1
    (implies (and (load-store-unit-p LSU)
                  (MA-state-p MA)
                  (MA-input-p sigs))
             (write-buffer-p (update-wbuf1 LSU MA sigs))))
(in-theory (disable update-wbuf1))

; The next state of write buffer 0.  If write buffer 0 is empty or
; a write operation is released in this cycle, write buffer 0 receives
; the trickle-down request from write buffer 1, represented by wbuf1-output.
; Otherwise, it keeps the original request stored in write buffer 0.
(defun step-wbuf0 (LSU MA sigs)
  (declare (xargs :guard (and (load-store-unit-p LSU)
                              (MA-state-p MA) (MA-input-p sigs))))
  (b-if (b-orc1 (wbuf-valid? (LSU-wbuf0 LSU)) (release-wbuf0? LSU sigs))
        (wbuf1-output LSU MA sigs)
        (update-wbuf0 LSU MA sigs)))

(defthm write-buffer-p-step-wbuf0
    (implies (and (load-store-unit-p LSU)
                  (MA-state-p MA)
                  (MA-input-p sigs))
             (write-buffer-p (step-wbuf0 LSU MA sigs))))
(in-theory (disable step-wbuf0))

; The next state of write buffer 1.  There are two cases where write
; buffer 1 accepts a new write request; One case is that the write
; buffer is full, a write is released this cycle and a new write is
; issued.  The other case is write buffer 0 contains a write request
; which is not released in this cycle, and a new write is issued.
(defun step-wbuf1 (LSU MA sigs)
  (declare (xargs :guard (and (load-store-unit-p LSU)
                              (MA-state-p MA) (MA-input-p sigs))))
  (let ((wbuf0 (LSU-wbuf0 LSU)) (wbuf1 (LSU-wbuf1 LSU)))
    (b-if (b-ior (b-and (wbuf-valid? wbuf1) (release-wbuf0? LSU sigs))
                 (bs-and (b-not (wbuf-valid? wbuf1))
                         (wbuf-valid? wbuf0)
                         (b-not (release-wbuf0? LSU sigs))))
          (issued-write LSU MA sigs)
          (update-wbuf1 LSU MA sigs))))

(defthm write-buffer-p-step-wbuf1
```

```
    (implies (and (load-store-unit-p LSU)
                  (MA-state-p MA)
                  (MA-input-p sigs))
             (write-buffer-p (step-wbuf1 LSU MA sigs))))
(in-theory (disable step-wbuf1))

; The next state of the LSU.
(defun step-LSU (MA sigs)
  (declare (xargs :guard (and (MA-state-p MA) (MA-input-p sigs))))
  (let ((LSU (MA-LSU MA)))
    (load-store-unit (step-rs1-head? LSU MA sigs)
                     (step-LSU-RS0 LSU MA sigs)
                     (step-LSU-RS1 LSU MA sigs)
                     (step-rbuf LSU MA sigs)
                     (step-wbuf0 LSU MA sigs)
                     (step-wbuf1 LSU MA sigs)
                     (step-LSU-lch LSU MA sigs))))

(defthm LSU-p-step-LSU
    (implies (and (MA-state-p MA) (MA-input-p sigs))
             (load-store-unit-p (step-LSU MA sigs))))
(in-theory (disable step-LSU))

; The next state of the memory.
(defun step-mem (MA sigs)
  (declare (xargs :guard (and (MA-state-p MA) (MA-input-p sigs))))
  (let ((mem (MA-mem MA)) (LSU (MA-LSU MA)))
    (b-if (release-wbuf0? LSU sigs)
          (write-mem (wbuf-val (LSU-wbuf0 LSU))
                     (wbuf-addr (LSU-wbuf0 LSU))
                     mem)
          mem)))

(defthm mem-p-step-mem
    (implies (and (MA-state-p MA) (MA-input-p sigs))
             (mem-p (step-mem MA sigs))))
(in-theory (disable step-mem))

(deflabel end-MA-step-functions)

;; The next state function of the pipelined machine.
;; It takes a pipeline state and returns the state one clock cycle later.
(defun MA-step (MA sigs)
  (declare (xargs :guard (and (MA-state-p MA) (MA-input-p sigs))))
  (MA-state (step-pc MA sigs)
            (step-RF MA)
            (step-SRF MA sigs)
            (step-IFU MA sigs)
            (step-DQ MA sigs)
            (step-ROB MA sigs)
            (step-IU MA sigs)
            (step-MU MA sigs)
            (step-BU MA sigs)
            (step-LSU MA sigs)
            (step-mem MA sigs)))

(defthm MA-state-p-MA-step
    (implies (and (MA-state-p MA) (MA-input-p sigs))
             (MA-state-p (MA-step MA sigs))))

(in-theory (disable MA-step))
```

```
;; MA-stepn runs the pipelined machine n cycles from the initial
;; state MA.
(defun MA-stepn (MA sigs-lst n)
  (declare (xargs :guard (and (MA-state-p MA) (integerp n) (>= n 0)
                              (MA-input-listp sigs-lst)
                              (<= n (len sigs-lst)))
                  :verify-guards nil))
  (if (zp n)
      MA
      (MA-stepn (MA-step MA (car sigs-lst)) (cdr sigs-lst) (1- n))))

(verify-guards MA-stepn)
(defthm MA-state-p-MA-stepn
    (implies (and (MA-state-p MA) (MA-input-listp sigs-lst)
                  (<= n (len sigs-lst)))
             (MA-state-p (MA-stepn MA sigs-lst n))))

(deflabel end-MA-def)

(deflabel begin-MA-flushed-def)
; The definition of a flushed state.  We consider the pipeline
; is flushed, when all latches contain no busy flag, and no external
; interrupt is on pending.
(defun IFU-empty? (IFU)
  (declare (xargs :guard (IFU-p IFU)))
  (b-not (IFU-valid? IFU)))

(defun reg-ref-empty? (idx DQ)
  (declare (xargs :guard (and (DQ-p DQ) (rname-p idx))))
  (b-not (reg-ref-wait? (reg-tbl-nth idx (DQ-reg-tbl DQ)))))

(defun reg-tbl-empty-under? (idx DQ)
  (declare (xargs :guard (and (DQ-p DQ) (integerp idx)
                              (<= 0 idx) (<= idx *num-regs*))
                  :verify-guards nil))
  (if (zp idx)
      1
      (b-and (reg-ref-empty? (1- idx) DQ)
             (reg-tbl-empty-under? (1- idx) DQ))))

(defthm bitp-reg-tbl-empty-under
    (bitp (reg-tbl-empty-under? idx DQ)))

(verify-guards reg-tbl-empty-under?
               :hints (("goal" :in-theory (enable rname-p
                                                  unsigned-byte-p))))

(defun reg-tbl-empty? (DQ)
  (declare (xargs :guard (DQ-p DQ)))
  (reg-tbl-empty-under? *num-regs* DQ))

(defun sreg-ref-empty? (idx DQ)
  (declare (xargs :guard (and (srname-p idx) (DQ-p DQ))
                  :guard-hints (("goal" :in-theory (enable SRNAME-P)))))
  (b-not (reg-ref-wait? (sreg-tbl-nth idx (DQ-sreg-tbl DQ)))))

(defun sreg-tbl-empty? (DQ)
  (declare (xargs :guard (DQ-p DQ)))
  (b-and (sreg-ref-empty? 0 DQ) (sreg-ref-empty? 1 DQ)))

(defun DQ-empty? (DQ)
  (declare (xargs :guard (DQ-p DQ)))
```

```
  (bs-and (b-not (DE-valid? (DQ-DE0 DQ)))
          (b-not (DE-valid? (DQ-DE1 DQ)))
          (b-not (DE-valid? (DQ-DE2 DQ)))
          (b-not (DE-valid? (DQ-DE3 DQ)))
          (reg-tbl-empty? DQ)
          (sreg-tbl-empty? DQ)))

(defun IU-empty? (IU)
  (declare (xargs :guard (integer-unit-p IU)))
  (bs-and (b-not (RS-valid? (IU-rs0 IU)))
          (b-not (RS-valid? (IU-rs1 IU)))))

(defun MU-empty? (MU)
  (declare (xargs :guard (mult-unit-p MU)))
  (bs-and (b-not (RS-valid? (MU-rs0 MU)))
          (b-not (RS-valid? (MU-rs1 MU)))
          (b-not (MU-latch1-valid? (MU-lch1 MU)))
          (b-not (MU-latch2-valid? (MU-lch2 MU)))))

(defun BU-empty? (BU)
  (declare (xargs :guard (branch-unit-p BU)))
  (bs-and (b-not (BU-RS-valid? (BU-rs0 BU)))
          (b-not (BU-RS-valid? (BU-rs1 BU)))))

(defun LSU-empty? (LSU)
  (declare (xargs :guard (load-store-unit-p LSU)))
  (bs-and (b-not (LSU-RS-valid? (LSU-rs0 LSU)))
          (b-not (LSU-RS-valid? (LSU-rs1 LSU)))
          (b-not (LSU-latch-valid? (LSU-lch LSU)))
          (b-not (wbuf-valid? (LSU-wbuf0 LSU)))
          (b-not (wbuf-valid? (LSU-wbuf1 LSU)))
          (b-not (rbuf-valid? (LSU-rbuf LSU)))))

(defun exintr-flag? (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (ROB-exintr? (MA-ROB MA)))

; The flushed state of the pipelined design.
(defun MA-flushed? (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (bs-and (IFU-empty? (MA-IFU MA))
          (DQ-empty? (MA-DQ MA))
          (ROB-empty? (MA-ROB MA))
          (ROB-entries-empty? (MA-ROB MA))
          (IU-empty? (MA-IU MA))
          (MU-empty? (MA-MU MA))
          (BU-empty? (MA-BU MA))
          (LSU-empty? (MA-LSU MA))
          (b-not (exintr-flag? MA))))

(defthm bitp-MA-flushed? (bitp (MA-flushed? MA)))

(defun flushed-p (MA) (b1p (MA-flushed? MA)))

(deflabel end-MA-flushed-def)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Number of Commits
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun num-commits (MA sigs-lst n)
  (declare (xargs :guard (and (MA-state-p MA) (MA-input-listp sigs-lst)
                              (integerp n) (<= 0 n))
```

```
                :measure (nfix n)))
  (if (or (zp n) (endp sigs-lst))
      0
      (b-if (b-ior (commit-inst? MA)
                   (ex-intr? MA (car sigs-lst)))
            (1+ (num-commits (MA-step MA (car sigs-lst))
                             (cdr sigs-lst) (1- n)))
            (num-commits (MA-step MA (car sigs-lst)) (cdr sigs-lst) (1- n)))))

(deftheory MA-state-def
    (set-difference-theories (function-theory 'end-MA-state)
                             (function-theory 'begin-MA-state)))
(deftheory MA-step-functions
    (set-difference-theories (function-theory 'end-MA-step-functions)
                             (function-theory 'begin-MA-step-functions)))

(deftheory MA-def
    (set-difference-theories (function-theory 'end-MA-def)
                             (function-theory 'begin-MA-def)))

(deftheory low-level-MA-def
    (set-difference-theories (theory 'MA-def) '(ma-step MA-stepn)))

(deftheory MA-def-all
    (union-theories (theory 'MA-def)
                    (theory 'MA-state-def)))

(deftheory MA-flushed-def
    (set-difference-theories (universal-theory 'end-MA-flushed-def)
                             (universal-theory 'begin-MA-flushed-def)))

(in-theory (disable MA-flushed? IFU-empty? DQ-empty? IU-empty?
                    MU-empty? BU-empty? LSU-empty? exintr-flag?
                    reg-tbl-empty? sreg-tbl-empty? reg-ref-empty?))

(in-theory (disable MA-def))

#|

Here is a simple example program for our pipelined machine.

Our program calculates the factorial of the number at address #x800 and stores
it at address #x801.

Initial memory setting:

#x0: ST R0,(#x50)
#x1: LD R0,(#x3)
#x2: BZ R0, 0
#x3: 0

#x10: ST R0,(#x50)
#x11: LD R0,(#x13)
#x12: BZ R0, 0
#x13: 0

#x20: ST R0,(#x50)
#x21: LD R0,(#x23)
#x22: BZ R0, 0
#x23: 0

#x30: ST R0,(#x50)
```

```
#x31: LD R0,(#x33)
#x32: BZ R0, 0
#x33: 0

#x60: 0
#x61: 1
#x62: 2
#x63: -1

#x70: #x400
#x71: #x800

#x100: LD R15,(#x70) ; program base
#x101: LD R14,(#x71) ; data base
#x102: LD R0, (#x60) ; 0
#x103: LD R1, (#x61) ; 1
#x104: LD R2, (#x62) ; 2
#x105: LD R3, (#x63) ; -1
#x106: MTSR SR0,R15
#x107: MTSR SR1,R0
#x108: RFEH

Initial memory image:
#x400     LD R5,(R14+R0) ; R5 holds counter
#x401     ADD R6, R0, R1     ; R6 holds factorial. Initially 1.
Loop:
#x402:    Mul R6, R6, R5  ; counter * fact -> fact
#x403:    ADD R5, R5, R3  ; decrement fact
#x404:    BZ  R5, Exit; if counter is zero, exit
#x405:    BZ  R0, Loop ; always jump to loop
EXIT:
#x406: ST  R6, (R14+R1)
#x407: SYNC
#x408: Trap

#x800: 5
#x801: 0
#x802: 5 ; Offset to Loop
#x803: 9 ; Offset to Exit

How to run the program:
1. certify and compile all the proof scripts.
   (You may skip this, but the execution will be slow.)
2. Run ACL2.
3. Type command '(ld "MA-def.lisp")'.
4  Evaluate expressions below and set initial state MA.
5. You can run the MA machine for one cycle by
   (MA-step (@ MA) (MA-input-p 0 1 1)).
   You can also run the machine for multiple cycles with MA-stepn.
   For instance, if you want to run the machine 15 cycles, type:
     (assign sigs-list (make-list 15 :initial-element (MA-input 0 1 1)))
     (MA-stepn (@ MA) 15 sigs-list).

6  Following macro may be useful to evaluate "expr" and set it to variable
   MA, without printing the state of memory.

; Evaluate expression expr and set the result to MA.
(defmacro eval-set-print-MA (MA expr)
  `(pprogn (f-put-global ',MA ,expr state)
           (mv nil
             (list (MA-pc (f-get-global ',MA state))
                   (MA-RF (f-get-global ',MA state))
```

```
                  (MA-SRF (f-get-global ',MA state))
                  (MA-DQ (f-get-global ',MA state))
                  (MA-ROB (f-get-global ',MA state))
                  (MA-IU (f-get-global ',MA state))
                  (MA-MU (f-get-global ',MA state))
                  (MA-BU (f-get-global ',MA state))
                  (MA-LSU (f-get-global ',MA state)))
              state)))

; Function to be used in MA-step-seq
(defun make-MA-step-seq (sigs seq)
  (if (endp seq) nil
      (if (endp (cdr seq)) nil
          (cons `(f-put-global ',(cadr seq) (MA-step (@ ,(car seq)) ,sigs)
                               state)
                (make-MA-step-seq sigs (cdr seq))))))

; Given an MA inputs and sequence of symbols, and execute MA-step one
; at a time and assigns its result to the symbol in the sequence.  For
; instance, (MA-step-seq (@ sigs) MA0 MA1 MA2 MA3) assigns the result
; of applying MA-step to MA0 to MA1, the result of applying MA-step to
; MA1 to MA2, and so on.
(defmacro MA-step-seq (sigs &rest seq)
  (if (endp seq) nil
      `(pprogn
        ,@(make-MA-step-seq sigs seq)
        (mv nil
         (list (MA-pc (f-get-global ',(car (last seq)) state))
          (MA-RF (f-get-global ',(car (last seq)) state))
          (MA-SRF (f-get-global ',(car (last seq)) state))
          (MA-ROB (f-get-global ',(car (last seq)) state)))
          state))))

(defmacro pr-MA (MA)
  `(list (MA-pc (@ ,MA)) (MA-RF (@ ,MA)) (MA-SRF (@ ,MA))
    (MA-IFU (@ ,MA)) (MA-DQ (@ ,MA)) (MA-ROB (@ ,MA)) (MA-IU (@ ,MA))
    (MA-MU (@ ,MA)) (MA-BU (@ ,MA)) (MA-LSU (@ ,MA))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Initial State Setting
(progn
(assign RF '(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
(assign SRF (SRF 1 0 0))

(assign IFU (IFU 0 0 0 0))
(assign DE (dispatch-entry 0 0 0 0 0 0 0 0 0))
(assign reg-MA (reg-ref 0 0))
(assign reg-tbl (make-list *num-regs* :initial-element (@ reg-MA)))
(assign sreg-tbl (sreg-tbl (@ reg-MA) (@ reg-MA)))
(assign DQ (DQ (@ DE) (@ DE) (@ DE) (@ DE) (@ reg-tbl) (@ sreg-tbl)))

(assign ROBE (ROB-entry 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
(assign entries (make-list *rob-size* :initial-element (@ ROBE)))
(assign ROB (ROB 0 0 0 0 (@ entries)))

(assign IU (integer-unit (RS 0 0 0 0 0 0 0 0 0) (RS 0 0 0 0 0 0 0 0 0)))
(assign MU (mult-unit (RS 0 0 0 0 0 0 0 0 0) (RS 0 0 0 0 0 0 0 0 0)
                      (MU-latch1 0 0 0) (MU-latch2 0 0 0)))
(assign BU (branch-unit (BU-RS 0 0 0 0 0) (BU-RS 0 0 0 0 0)))
(assign LSU (load-store-unit 0
                             (LSU-RS 0 0 0 0  0 0 0 0 0 0 0 0 0 0)
                             (LSU-RS 0 0 0 0  0 0 0 0 0 0 0 0 0 0)
```

```
                          (read-buffer 0 0 0 0 0)
                          (write-buffer 0 0 0 0 0 0)
                          (write-buffer 0 0 0 0 0 0)
                          (LSU-latch 0 0 0 0)))
(assign mem-alist '(
; Exception Handler
(#x0 . #x7050) ; ST R0,(#x50)
(#x1 . #x6003) ; LD R0,(#x3)
(#x2 . #x2000) ; BZ R0, 0
(#x3 . 0)
; Exception Handler
(#x10 . #x7050) ; ST R0,(#x50)
(#x11 . #x6013) ; LD R0,(#x13)
(#x12 . #x2000) ; BZ R0, 0
(#x13 . 0)
; Exception Handler
(#x20 . #x7050) ; ST R0,(#x50)
(#x21 . #x6023) ; LD R0,(#x23)
(#x22 . #x2000) ; BZ R0, 0
(#x23 . 0)

; Exception Handler
(#x30 . #x7050) ; ST R0,(#x50)
(#x31 . #x6033) ; LD R0,(#x33)
(#x32 . #x2000) ; BZ R0, 0
(#x33 . 0)

; Kernel Data Section
(#x60 .  0)
(#x61 . 1)
(#x62 . 2)
(#x63 . #xFFFF) ; -1
(#x70 . #x400)
(#x71 . #x800)
; Kernel Dispatching code
(#x100 . #x6F70) ; LD R15,(#x70) ; program base
(#x101 . #x6E71) ;  LD R14,(#x71) ; data base
(#x102 . #x6060) ;  LD R0, (#x60) ; 0
(#x103 . #x6161) ;  LD R1, (#x61) ; 1
(#x104 . #x6262) ; LD R2, (#x62) ; 2
(#x105 . #x6363) ; LD R3, (#x63) ; -1
(#x106 . #xAF00) ;  MTSR SR0,R15
(#x107 . #xA010) ;  MTSR SR1,R0
(#x108 . #x8000) ; #x103: RFEH
; Program
(#x400 . #x35E0) ;  LD R5,(R14+R0) ; R5 holds counter
(#x401 . #x0601) ;  ADD R6, R0, R1     ; R6 holds factorial. Initially 1.
; Loop:
(#x402 . #x1665) ;  Mul R6, R6, R5  ; counter * fact -> fact
(#x403 . #x0553) ;  ADD R5, R5, R3  ; decrement fact
(#x404 . #x2502) ;  BZ  R5, Exit; if counter is zero, exit
(#x405 . #x20FD) ;  BZ  R0, Loop ; always jump to loop
; EXIT:
(#x406 . #x46E1) ; ST  R6, (R14+R1)
(#x407 . #x5000) ; SYNC
(#x408 . #xB000) ; Trap

; Data Section
(#x800 . 5)
(#x801 . 0)
(#x802 . 5) ; Offset to Loop
(#x803 . 9) ; Offset to Exit
```

```
))

(assign mem (set-page-mode *read-only* 1 (compress1 'mem *init-mem*)))
(assign mem (set-page-mode *read-write* 2 (@ mem)))
(assign mem (compress1 'mem (load-mem-alist (@ mem-alist) (@ mem))))
(assign MA (MA-state #x100 (@ RF) (@ SRF) (@ IFU) (@ DQ) (@ ROB)
                   (@ IU) (@ MU) (@ BU) (@ LSU) (@ mem)))

)
|#
```

D.3 Intermediate Abstraction

The definition of the MAETT intermediate abstraction. Please see Chapter 7 for discussions.

D.3.1 MAETT-def.lisp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MAETT-def.lisp:
; Author  Jun Sawada, University of Texas at Austin
;
;  This book defines the MAETT for our microarchitectural design.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(in-package "ACL2")

(include-book "ISA-def")
(include-book "MA2-def")
(include-book "utils")

(deflabel begin-MAETT-def)

;; Defining projection from MA states to ISA states.
(defun proj (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (ISA-state (MA-pc MA)
             (MA-RF MA)
             (MA-SRF MA)
             (MA-mem MA)))

(defthm ISA-state-p-proj
    (implies (MA-state-p MA) (ISA-state-p (proj MA))))

(defthm ISA-pc-proj
    (implies (MA-state-p MA) (equal (ISA-pc (proj MA)) (MA-pc MA))))

(defthm ISA-mem-proj
    (implies (MA-state-p MA)
             (equal (ISA-mem (proj MA)) (MA-mem MA))))

(defthm ISA-RF-proj
    (implies (MA-state-p MA)
             (equal (ISA-RF (proj MA)) (MA-RF MA))))
```

```
(defthm ISA-SRF-proj
    (implies (MA-state-p MA)
             (equal (ISA-SRF (proj MA)) (MA-SRF MA))))

(in-theory (disable proj))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Here begins the definition of MAETT
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deflabel begin-MA-stg-def)

(defun IFU-stg-p (stg)
  (declare (xargs :guard t))
  (equal stg '(IFU)))

(defun DQ-stg-p (stg)
  (declare (xargs :guard t))
  (or (equal stg '(DQ 0))
      (equal stg '(DQ 1))
      (equal stg '(DQ 2))
      (equal stg '(DQ 3))))

(defun DQ-stg-idx (stg)
  (declare (xargs :guard (DQ-stg-p stg)))
  (cadr stg))

(defthm type-DQ-stg
    (implies (DQ-stg-p stg)
             (and (integerp (DQ-stg-idx stg))
                  (<= 0 (DQ-stg-idx stg))))
  :rule-classes ((:type-prescription)
                 (:rewrite :corollary
                           (implies (DQ-stg-p stg)
                                    (integerp (DQ-stg-idx stg))))
                 (:linear :corollary
                          (implies (DQ-stg-p stg)
                                   (<= 0 (DQ-stg-idx stg))))))

(defun IU-stg-p (stg)
  (declare (xargs :guard t))
  (or (equal stg '(IU RS0))
      (equal stg '(IU RS1))))

(defun BU-stg-p (stg)
  (declare (xargs :guard t))
  (or (equal stg '(BU RS0))
      (equal stg '(BU RS1))))

(defun MU-stg-p (stg)
  (declare (xargs :guard t))
  (or (equal stg '(MU RS0))
      (equal stg '(MU RS1))
      (equal stg '(MU lch1))
      (equal stg '(MU lch2))))

; Check if the stg represents a stage of an instruction in LSU.
; (LSU wbuf0) and (LSU wbuf1) represent a write at the write buffer with
; flag valid? on, but complete? and commit? off.  Then a write
; proceeds to a stage where it occupies both an write buffer entry and
; output latch.  These stages are represented by (LSU wbuf0 lch) or
; (LSU wbuf1 lch).
(defun LSU-stg-p (stg)
```

```
  (declare (xargs :guard t))
  (or (equal stg '(LSU RS0))
      (equal stg '(LSU RS1))
      (equal stg '(LSU rbuf))
      (equal stg '(LSU lch))
      (equal stg '(LSU wbuf0))
      (equal stg '(LSU wbuf1))
      (equal stg '(LSU wbuf0 lch))
      (equal stg '(LSU wbuf1 lch))))

; Execution stage.
(defun execute-stg-p (stg)
  (declare (xargs :guard t))
  (or (IU-stg-p stg)
      (MU-stg-p stg)
      (BU-stg-p stg)
      (LSU-stg-p stg)))

; An instruction has completed its execution and is waiting to be
; committed in the re-order buffer.  A memory write operation is not
; performed until the corresponding instruction commits.  Meanwhile
; the write buffer contains the corresponding write request. The stage
; of a store instruction which is at the complete stage is either
; (complete wbuf0) or (complete wbuf1) depending on the write buffer
; entry storing the corresponding write request.
(defun complete-stg-p (stg)
  (declare (xargs :guard t))
  (or (equal stg '(complete))
      (equal stg '(complete wbuf0))
      (equal stg '(complete wbuf1))))

; Commit stage only happens for a store instruction.  An instruction
; at this stage does not have a corresponding ROB entry anymore.
; The write is waiting to be completed in the write buffer.
(defun commit-stg-p (stg)
  (declare (xargs :guard t))
  (or (equal stg '(commit wbuf0))
      (equal stg '(commit wbuf1))))

; Retired.  Everything is over.
(defun retire-stg-p (stg)
  (declare (xargs :guard t))
  (equal stg '(RETIRE)))

(defun stage-p (stg)
  (declare (xargs :guard t))
  (or (IFU-stg-p stg)
      (DQ-stg-p stg)
      (execute-stg-p stg)
      (complete-stg-p stg)
      (commit-stg-p stg)
      (retire-stg-p stg)))

(defun RS-stg-p (stg)
  (or (equal stg '(IU RS0))
      (equal stg '(IU RS1))
      (equal stg '(BU RS0))
      (equal stg '(BU RS1))
      (equal stg '(MU RS0))
      (equal stg '(MU RS1))
      (equal stg '(LSU RS0))
      (equal stg '(LSU RS1))))
```

```
(defun wbuf-stg-p (stg)
  (declare (xargs :guard t))
  (or (equal stg '(LSU wbuf0))
      (equal stg '(LSU wbuf1))
      (equal stg '(LSU wbuf0 lch))
      (equal stg '(LSU wbuf1 lch))
      (equal stg '(complete wbuf0))
      (equal stg '(complete wbuf1))
      (equal stg '(commit wbuf0))
      (equal stg '(commit wbuf1))))

(defun wbuf0-stg-p (stg)
  (declare (xargs :guard t))
  (or (equal stg '(LSU wbuf0))
      (equal stg '(LSU wbuf0 lch))
      (equal stg '(complete wbuf0))
      (equal stg '(commit wbuf0))))

(defun wbuf1-stg-p (stg)
  (declare (xargs :guard t))
  (or (equal stg '(LSU wbuf1))
      (equal stg '(LSU wbuf1 lch))
      (equal stg '(complete wbuf1))
      (equal stg '(commit wbuf1))))

(deflabel end-MA-stg-def)

#|
Following is the definition of INST record.

ID:  Identity of i
modify?:  Whether i is modified by earlier STORE
          instructions.  In a well-formed MAETT, this flag
          is sticky in the sense that modify? flag of i is 1 if that
          of the preceding instruction is 1.
specultv?: Whether the instruction is speculatively executed. This
          flag is also sticky.
br-predict?: If i is a branch instruction, this field records the output of
          branch prediction invoked by i.
exintr?: Record whether the i is interrupted by an external interrupt.
stg: The current pipeline stage of i.
tag: The index to the ROB entry in which i is stored.
     The same value is used for the tag of the instruction.
pre-ISA: The correct ISA state before executing i.
post-ISA: The correct ISA state after executing i.
|#

(defstructure INST
  (ID (:assert (naturalp ID) :type-prescription))
  (modified? (:assert (bitp modified?) :rewrite))
  (specultv? (:assert (bitp specultv?) :rewrite))
  (first-modified? (:assert (bitp first-modified?) :rewrite))
  (br-predict? (:assert (bitp br-predict?) :rewrite))
  (exintr? (:assert (bitp exintr?) :rewrite))
  (stg (:assert (stage-p stg) :rewrite))
  (tag (:assert (rob-index-p tag) :rewrite
                (:type-prescription (and (integerp tag) (<= 0 tag)))
                (:rewrite (acl2-numberp tag))))
  (pre-ISA (:assert (ISA-state-p pre-ISA) :rewrite))
  (post-ISA (:assert (ISA-state-p post-ISA) :rewrite))
  (:options :guards))
```

```
(deflist INST-listp (l)
  (declare (xargs :guard t))
  INST-p)

; Micro-architectural Execution Trace consists of several data items
; init-ISA:  the ISA state corresponding to the initial state of MA
; new-ID:  An ID which is incremented every cycle.
; dq-len: The number of valid entries in the dispatch queue.
; wb-len: The number of valid entries in the write buffer.
; rob-flg: The flag used to implement a circular buffer for ROB.
; rob-head: The ROB index to the first instruction in ROB.
; rob-tail: The ROB index to the last instruction in ROB.
; trace: The list of traced instructions.
(defstructure MAETT
  (init-ISA (:assert (ISA-state-p init-ISA) :rewrite))
  (new-ID (:assert (naturalp new-ID) :type-prescription))
  (dq-len (:assert (naturalp dq-len) :type-prescription))
  (wb-len (:assert (naturalp wb-len) :type-prescription))
  (rob-flg (:assert (bitp rob-flg) :rewrite))
  (rob-head (:assert (rob-index-p rob-head) :rewrite
                     (:type-prescription (integerp rob-head))))
  (rob-tail (:assert (rob-index-p rob-tail) :rewrite
                     (:type-prescription (integerp rob-tail))))
  (trace (:assert (INST-listp trace) :rewrite))
  (:options :guards (:conc-name MT-)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Definition of INST functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun dispatched-p (i)
  (declare (xargs :guard (inst-p i)))
  (or (execute-stg-p (INST-stg i)) (complete-stg-p (INST-stg i))
      (commit-stg-p (INST-stg i)) (retire-stg-p (INST-stg i))))

(defun committed-p (i)
  (declare (xargs :guard (inst-p i)))
  (or (commit-stg-p (INST-stg i)) (retire-stg-p (INST-stg i))))

(deflabel begin-of-MT-def)

(deflabel begin-INST-function-def)

(defun INST-pc (i)
  (declare (xargs :guard (INST-p i)))
  (ISA-pc (INST-pre-ISA i)))

(defthm addr-INST-pc
    (implies (INST-p i) (addr-p (INST-pc i))))

(defun INST-RF (i)
  (declare (xargs :guard (INST-p i)))
  (ISA-RF (INST-pre-ISA i)))

(defthm RF-p-INST-RF
    (implies (INST-p i) (RF-p (INST-RF i))))

(defun INST-SRF (i)
  (declare (xargs :guard (INST-p i)))
  (ISA-SRF (INST-pre-ISA i)))

(defthm SRF-p-INST-SRF
```

```
    (implies (INST-p i) (SRF-p (INST-SRF i))))

(defun INST-mem (i)
  (declare (xargs :guard (INST-p i)))
  (ISA-mem (INST-pre-ISA i)))

(defthm mem-p-INST-mem
    (implies (INST-p i) (mem-p (INST-mem i))))

(defun INST-su (i)
  (declare (xargs :guard (INST-p i)))
  (SRF-su (ISA-SRF (INST-pre-ISA i))))

(defthm bitp-INST-su
    (implies (INST-p i) (bitp (INST-su i))))

(defun INST-word (i)
  (declare (xargs :guard (INST-p i)))
  (read-mem (INST-pc i) (INST-mem i)))

(defthm word-p-INST-word
    (implies (INST-p i) (word-p (INST-word i))))

(defthm integerp-INST-word
    (implies (INST-p i) (integerp (INST-word i)))
  :rule-classes :type-prescription)

(defun INST-opcode (i)
  (declare (xargs :guard (INST-p i)))
  (opcode (INST-word i)))

(defthm opcd-p-INST-opcode
    (opcd-p (INST-opcode i)))

(defun INST-ra (i)
  (declare (xargs :guard (INST-p i)))
  (ra (INST-word i)))

(defthm rname-p-INST-ra
    (rname-p (INST-ra i)))

(defun INST-rb (i)
  (declare (xargs :guard (INST-p i)))
  (rb (INST-word i)))

(defthm rname-p-INST-rb
    (rname-p (INST-rb i)))

(defun INST-rc (i)
  (declare (xargs :guard (INST-p i)))
  (rc (INST-word i)))

(defthm rname-p-INST-rc
    (rname-p (INST-rc i)))

(defun INST-im (i)
  (declare (xargs :guard (INST-p i)))
  (im (INST-word i)))

(defthm immediate-p-INST-im
    (immediate-p (INST-im i)))
```

```
(defun INST-fetch-error? (i)
  (declare (xargs :guard (INST-p i)))
  (let ((s (INST-pre-ISA i)))
    (read-error? (ISA-pc s) (ISA-mem s) (SRF-su (ISA-SRF s)))))

(defthm bitp-INST-fetch-error? (bitp (INST-fetch-error? i)))

(defun INST-decode-error? (i)
  (declare (xargs :guard (INST-p i)))
  (let ((s (INST-pre-ISA i)))
    (decode-illegal-inst? (opcode (INST-word i))
                          (SRF-su (ISA-SRF s))
                          (INST-ra i))))

(defthm bitp-INST-decode-error? (bitp (INST-decode-error? i)))

(defun INST-load-error? (i)
  (declare (xargs :guard (INST-p i)))
  (let ((s (INST-pre-ISA i))
        (inst (INST-word i)))
    (let ((su (SRF-su (ISA-SRF s)))
          (mem (ISA-mem s))
          (RF (ISA-RF s)))
      (b-if (bv-eqv *opcode-size* (opcode inst) 6)
            (read-error? (addr (im inst)) mem su)
      (b-if (bv-eqv *opcode-size* (opcode inst) 3)
            (read-error? (addr (+ (read-reg (ra inst) RF)
                                  (read-reg (rb inst) RF)))
                         mem su)
            0)))))

(defthm bitp-INST-load-error? (bitp (INST-load-error? i)))
(in-theory (disable INST-load-error?))

(defun INST-store-error? (i)
  (declare (xargs :guard (INST-p i)))
  (let ((s (INST-pre-ISA i))
        (inst (INST-word i)))
    (let ((su (SRF-su (ISA-SRF s)))
          (mem (ISA-mem s))
          (RF (ISA-RF s)))
      (b-if (bv-eqv *opcode-size* (opcode inst) 7)
            (write-error? (addr (im inst)) mem su)
      (b-if (bv-eqv *opcode-size* (opcode inst) 4)
            (write-error? (addr (+ (read-reg (ra inst) RF)
                                   (read-reg (rb inst) RF)))
                          mem su)
            0)))))

(defthm bitp-INST-store-error? (bitp (INST-store-error? i)))
(in-theory (disable INST-store-error?))

(defun INST-data-access-error? (i)
  (declare (xargs :guard (INST-p i)))
  (b-ior (INST-load-error? i)
         (INST-store-error? i)))

(defthm bitp-INST-data-access-error? (bitp (INST-data-access-error? i)))

(defun INST-excpt? (i)
  (declare (xargs :guard (INST-p i)))
  (bs-ior (INST-fetch-error? i)
```

```
          (INST-decode-error? i)
          (INST-data-access-error? i)))

(defthm bitp-INST-excpt (bitp (INST-excpt? i)))

(defun INST-fetch-error-detected-p (i)
  (declare (xargs :guard (INST-p i)))
  (let ((s (INST-pre-ISA i)))
    (b1p (read-error? (ISA-pc s) (ISA-mem s) (SRF-su (ISA-SRF s))))))

(defun INST-cntlv (i)
  (declare (xargs :guard (INST-p i)))
  (decode (INST-opcode i) (INST-br-predict? i)))

(defthm cntlv-p-INST-cntlv
    (implies (INST-p i) (cntlv-p (INST-cntlv i))))

(defun inst-br-target (i)
  (declare (xargs :guard (INST-p i)))
  (addr (+ (INST-pc i)
           (logextu *addr-size* *immediate-size* (im (INST-word i))))))

(defthm addr-p-inst-br-target
    (addr-p (inst-br-target i)))

(defun INST-load-addr (i)
  (declare (xargs :guard (INST-p i)))
  (cond ((equal (INST-opcode i) 3)
         (addr (+ (read-reg (INST-ra i) (ISA-RF (INST-pre-ISA i)))
                  (read-reg (INST-rb i) (ISA-RF (INST-pre-ISA i))))))
        ((equal (INST-opcode i) 6) (addr (INST-im i)))
        (t 0)))

(defthm addr-p-INST-load-addr
    (addr-p (INST-load-addr i)))

(defun INST-store-addr (i)
  (declare (xargs :guard (INST-p i)))
  (cond ((equal (INST-opcode i) 4)
         (addr (+ (read-reg (INST-ra i) (ISA-RF (INST-pre-ISA i)))
                  (read-reg (INST-rb i) (ISA-RF (INST-pre-ISA i))))))
        ((equal (INST-opcode i) 7) (addr (INST-im i)))
        (t 0)))

(defthm addr-p-INST-store-addr
    (addr-p (INST-store-addr i)))

(defun INST-src-val1 (i)
  (declare (xargs :guard (INST-p i)))
  (let ((op (INST-opcode i))
        (ra (INST-ra i))
        (rc (INST-rc i))
        (im (INST-im i))
        (RF (ISA-RF (INST-pre-ISA i)))
        (SRF (ISA-SRF (INST-pre-ISA i))))
    (cond ((or (equal op 0) (equal op 1) (equal op 3) (equal op 4))
           (read-reg ra RF))
          ((or (equal op 6) (equal op 7))
           (word im))
          ((or (equal op 2) (equal op 10))
           (read-reg rc RF))
          ((equal op 9)
```

```
          (read-sreg ra SRF))
         (t 0))))

(defthm word-p-INST-src-val1
    (implies (INST-p i) (word-p (INST-src-val1 i))))

(defun INST-src-val2 (i)
  (declare (xargs :guard (INST-p i)))
  (read-reg (INST-rb i) (ISA-RF (INST-pre-ISA i))))

(defthm word-p-INST-src-val2
    (implies (INST-p i) (word-p (INST-src-val2 i))))

(defun INST-src-val3 (i)
  (declare (xargs :guard (INST-p i)))
  (read-reg (INST-rc i) (ISA-RF (INST-pre-ISA i))))

(defthm word-p-INST-src-val3
    (implies (INST-p i) (word-p (INST-src-val3 i))))

(defun INST-ADD-dest-val (i)
  (declare (xargs :guard (INST-p i)))
  (word (+ (INST-src-val1 i) (INST-src-val2 i))))

(defthm word-p-INST-ADD-dest-val
    (word-p (INST-ADD-dest-val i)))

(defun INST-MULT-dest-val (i)
  (declare (xargs :guard (INST-p i)))
  (word (* (INST-src-val1 i) (INST-src-val2 i))))

(defthm word-p-INST-MULT-dest-val
    (word-p (INST-MULT-dest-val i)))

(defun INST-LD-dest-val (i)
  (declare (xargs :guard (INST-p i)))
  (read-mem (addr (+ (INST-src-val1 i) (INST-src-val2 i)))
            (ISA-mem (INST-pre-ISA i))))

(defthm word-p-INST-LD-dest-val
    (implies (INST-p i) (word-p (INST-LD-dest-val i))))

(defun INST-LD-im-dest-val (i)
  (declare (xargs :guard (INST-p i)))
  (read-mem (addr (INST-src-val1 i)) (ISA-mem (INST-pre-ISA i))))

(defthm word-p-INST-LD-im-dest-val
    (implies (INST-p i) (word-p (INST-LD-im-dest-val i))))

(defun INST-MFSR-dest-val (i)
  (declare (xargs :guard (INST-p i)))
  (INST-src-val1 i))

(defthm word-p-INST-MFSR-dest-val
    (implies (INST-p i) (word-p (INST-MFSR-dest-val i))))

(defun INST-MTSR-dest-val (i)
  (declare (xargs :guard (INST-p i)))
  (INST-src-val1 i))

(defthm word-p-INST-MTSR-dest-val
    (implies (INST-p i) (word-p (INST-MTSR-dest-val i))))
```

```
; This is true for any instruction i which writes back its value to
; the register file or the special register file.
; INST-writeback-p is true if and only if INST-wb? is 1.
(defun INST-writeback-p (i)
  (declare (xargs :guard (INST-p i)))
  (let ((op (opcode (INST-word i))))
    (or (equal op 0) (equal op 1) (equal op 3) (equal op 6)
        (equal op 9) (equal op 10))))

(defun INST-dest-val (i)
  (declare (xargs :guard (INST-p i)))
  (let ((op (opcode (INST-word i))))
    (cond ((equal op 0)
           (INST-ADD-dest-val i))
          ((equal op 1)
           (INST-MULT-dest-val i))
          ((equal op 3)
           (INST-LD-dest-val i))
          ((equal op 6)
           (INST-LD-im-dest-val i))
          ((equal op 9)
           (INST-MFSR-dest-val i))
          ((equal op 10)
           (INST-MTSR-dest-val i))
          (t 0))))

(defthm word-p-INST-dest-val
    (implies (INST-p i) (word-p (INST-dest-val i))))

(defun INST-dest-reg (i)
  (declare (xargs :guard (INST-p i)))
  (let ((op (opcode (INST-word i))))
    (cond ((or (equal op 0) (equal op 1) (equal op 3) (equal op 6)
               (equal op 9))
           (INST-rc i))
          ((equal op 10) (INST-ra i))
          (t 0))))

(defthm rname-p-INST-dest-reg
    (rname-p (INST-dest-reg i)))

(defun INST-IU? (i)
  (declare (xargs :guard (INST-p i)))
  (logbit 0 (cntlv-exunit (INST-cntlv i))))

(defthm bitp-INST-IU?
    (bitp (INST-IU? i)))

(defun INST-MU? (i)
  (declare (xargs :guard (INST-p i)))
  (logbit 1 (cntlv-exunit (INST-cntlv i))))

(defthm bitp-INST-MU?
    (bitp (INST-MU? i)))

(defun INST-LSU? (i)
  (declare (xargs :guard (INST-p i)))
  (logbit 2 (cntlv-exunit (INST-cntlv i))))

(defthm bitp-INST-LSU?
    (bitp (INST-LSU? i)))
```

```
(defun INST-BU? (i)
  (declare (xargs :guard (INST-p i)))
  (logbit 3 (cntlv-exunit (INST-cntlv i))))

(defthm bitp-INST-BU?
    (bitp (INST-BU? i)))

(defun INST-no-unit? (i)
  (declare (xargs :guard (INST-p i)))
  (logbit 4 (cntlv-exunit (INST-cntlv i))))

(defthm bitp-INST-no-unit?
    (bitp (INST-no-unit? i)))

(defun INST-ld-st? (i)
  (declare (xargs :guard (INST-p i)))
  (cntlv-ld-st? (INST-cntlv i)))

(defthm bitp-INST-ld-st? (bitp (INST-ld-st? i)))

(defun INST-store? (i)
  (declare (xargs :guard (INST-p i)))
  (b-and (INST-LSU? i) (INST-ld-st? i)))

(defthm bitp-INST-store?
    (bitp (INST-store? i)))

(defun INST-load? (i)
  (declare (xargs :guard (INST-p i)))
  (b-andc2 (INST-LSU? i) (INST-ld-st? i)))

(defthm bitp-INST-load?
    (bitp (INST-load? i)))

(defun INST-wb? (i)
  (declare (xargs :guard (INST-p i)))
  (cntlv-wb? (INST-cntlv i)))

(defthm bitp-INST-wb? (bitp (INST-wb? i)))

; INST-wb-SRF? is used to indicate whether the writeback value
; is directed to the special register file or the general register file.
(defun INST-wb-sreg? (i)
  (declare (xargs :guard (INST-p i)))
  (cntlv-wb-sreg? (INST-cntlv i)))

(defthm bitp-INST-wb-sreg? (bitp (INST-wb-sreg? i)))

(defun INST-sync? (i)
  (declare (xargs :guard (INST-p i)))
  (cntlv-sync? (INST-cntlv i)))

(defthm bitp-INST-sync? (bitp (INST-sync? i)))

(defun INST-rfeh? (i)
  (declare (xargs :guard (INST-p i)))
  (cntlv-rfeh? (INST-cntlv i)))

(defthm bitp-INST-rfeh (bitp (INST-rfeh? i)))

(defun INST-branch-dest (i)
```

```
  (declare (xargs :guard (INST-p i)))
  (addr (+ (logextu *addr-size* *immediate-size* (INST-im i))
           (ISA-pc (INST-pre-ISA i)))))

(defthm addr-p-INST-branch-dest
    (addr-p (INST-branch-dest i)))

; Flags to indicate IU operation. If 0, add operation of source values
; val1 and val2.  If 1, IU simply returns an source operand val1.
(defun INST-IU-op? (i)
  (declare (xargs :guard (INST-p i)))
  (b-not (logbit 0 (cntlv-operand (INST-cntlv i)))))

(defthm bitp-INST-IU-op
    (bitp (INST-IU-op? i)))

; Operation flag for load store unit.  If this flag is on,
; Immediate value is used as the memory access address.
; Note:
; I forgot what op originally meant.  In this function, op
; probably meant "operands" rather than "operation". See the definition
; of LSU-RS in MA2-def.
(defun INST-LSU-op? (i)
  (declare (xargs :guard (INST-p i)))
  (logbit 1 (cntlv-operand (INST-cntlv i))))

(defthm bitp-INST-LSU-op
    (bitp (INST-LSU-op? i)))

; INST-commit? is true if instruction i commits at this cycle.
(defun INST-commit? (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (bs-and (commit-inst? MA)
          (if (complete-stg-p (INST-stg i)) 1 0)
          (bv-eqv *rob-index-size* (ROB-head (MA-ROB MA))
                  (INST-tag i))))

(defthm bitp-INST-commit (bitp (INST-commit? i MA)))

(defun INST-cause-jmp? (i MT MA sigs)
  (declare (xargs :guard (and (INST-p i) (MAETT-p MT)
                              (MA-state-p MA) (MA-input-p sigs))))
  (bs-and (if (complete-stg-p (INST-stg i)) 1 0)
          (bv-eqv *rob-index-size* (inst-tag i) (MT-rob-head MT))
          (bs-ior (commit-jmp? MA)
                  (enter-excpt? MA)
                  (leave-excpt? MA))))

(defthm bitp-INST-cause-jmp (bitp (INST-cause-jmp? i MT MA sigs)))

(defun INST-exintr-now? (i MA sigs)
  (declare (xargs :guard (and (INST-p i)
                              (MA-state-p MA) (MA-input-p sigs))))
  (if (or (IFU-stg-p (INST-stg i)) (DQ-stg-p (INST-stg i)))
      (bs-and (ROB-empty? (MA-ROB MA))
              (b-not (LSU-pending-writes? (MA-LSU MA)))
              (b-ior (ROB-exintr? (MA-ROB MA))
                     (MA-input-exintr sigs)))
      0))

(defthm bitp-INST-exintr-now (bitp (INST-exintr-now? i MA sigs)))
```

```
(defun INST-context-sync? (i)
  (declare (xargs :guard (INST-p i)))
  (b-andc1 (INST-excpt? i)
           (INST-sync? i)))

(defthm bitp-INST-context-sync (bitp (INST-context-sync? i)))

(defun INST-branch-taken? (i)
  (declare (xargs :guard (INST-p i)))
  (let ((s (INST-pre-ISA i)))
    (bv-eqv *word-size* (read-reg (rc (INST-word i)) (ISA-RF s)) 0)))

(defthm bitp-INST-branch-taken (bitp (INST-branch-taken? i)))

(defun INST-wrong-branch? (i)
  (declare (xargs :guard (INST-p i)))
  (bs-and (b-not (INST-excpt? i))
          (INST-BU? i)
          (cond ((IFU-stg-p (INST-stg i))
                 (INST-branch-taken? i))
                (t (b-xor (INST-branch-taken? i)
                          (INST-br-predict? i))))))

(defthm bitp-INST-wrong-branch? (bitp (INST-wrong-branch? i)))

(defun INST-start-specultv? (i)
  (declare (xargs :guard (INST-p i)))
  (if (committed-p i)
      0
      (bs-ior (INST-excpt? i)
              (INST-context-sync? i)
              (INST-wrong-branch? i))))

(defthm bitp-INST-start-specultv (bitp (INST-start-specultv? i)))

(defun INST-decode-error-detected-p (i)
  (declare (xargs :guard (INST-p i)))
  (let ((op (INST-opcode i)))
    (and (not (INST-fetch-error-detected-p i))
         (not (equal op 0))
         (not (equal op 1))
         (not (equal op 2))
         (not (equal op 3))
         (not (equal op 4))
         (not (equal op 5))
         (not (equal op 6))
         (not (equal op 7))
         (not (and (equal op 8) (b1p (INST-su i))))
         (not (and (equal op 9) (b1p (INST-su i))
                   (or (equal (INST-ra i) 0) (equal (INST-ra i) 1))))
         (not (and (equal op 10) (b1p (INST-su i))
                   (or (equal (INST-ra i) 0) (equal (INST-ra i) 1))))
         (not (IFU-stg-p (INST-stg i))))))

(defun INST-load-accs-error-detected-p (i)
  (declare (xargs :guard (INST-p i)))
  (let ((s (INST-pre-ISA i)) (op (INST-opcode i)))
    (and (not (INST-fetch-error-detected-p i))
         (or (equal op 3) (equal op 6))
         (b1p (read-error? (INST-load-addr i)
                           (ISA-mem s)
                           (SRF-su (ISA-SRF s))))
```

```
        (or (equal (INST-stg i) '(LSU lch))
            (complete-stg-p (INST-stg i))
            (commit-stg-p (INST-stg i))
            (retire-stg-p (INST-stg i))))))

(defun INST-store-accs-error-detected-p (i)
  (declare (xargs :guard (INST-p i)))
  (let ((s (INST-pre-ISA i)) (op (INST-opcode i)))
    (and (not (INST-fetch-error-detected-p i))
         (or (equal op 4) (equal op 7))
         (b1p (write-error? (INST-store-addr i)
                            (ISA-mem s)
                            (SRF-su (ISA-SRF s))))
         (or (equal (INST-stg i) '(LSU wbuf0 lch))
             (equal (INST-stg i) '(LSU wbuf1 lch))
             (complete-stg-p (INST-stg i))
             (commit-stg-p (INST-stg i))
             (retire-stg-p (INST-stg i))))))

(defun INST-data-accs-error-detected-p (i)
  (declare (xargs :guard (INST-p i)))
  (or (INST-load-accs-error-detected-p i)
      (INST-store-accs-error-detected-p i)))

(defun INST-excpt-detected-p (i)
  (declare (xargs :guard (INST-p i)))
  (or (INST-fetch-error-detected-p i)
      (INST-decode-error-detected-p i)
      (INST-data-accs-error-detected-p i)))

(defun INST-excpt-flags (i)
  (declare (xargs :guard (INST-p i)))
  (cond ((INST-fetch-error-detected-p i) #b101)
        ((INST-decode-error-detected-p i) #b100)
        ((INST-data-accs-error-detected-p i) #b110)
        (t 0)))

(defthm excpt-flags-p-INST-excpt-flags
    (excpt-flags-p (INST-excpt-flags i)))

(deflabel end-INST-function-def)

(defun trace-no-write-at (addr trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace)
      T
      (and (or (not (b1p (INST-store? (car trace))))
               (not (equal (INST-store-addr (car trace)) addr))
               (b1p (INST-excpt? (car trace)))
               (b1p (INST-exintr? (car trace))))
           (trace-no-write-at addr (cdr trace)))))

(defun MT-no-write-at (addr MT)
  (declare (xargs :guard (MAETT-p MT)))
  (trace-no-write-at addr (MT-trace MT)))

(defun fetched-inst (MT pre-ISA specultv? modified?)
  (declare (xargs :guard (and (ISA-state-p pre-ISA) (MAETT-p MT)
                              (bitp specultv?)
                              (bitp modified?))))
  (INST (MT-new-ID MT)
        (b-orc2 modified?
```

```
               (if (MT-no-write-at (ISA-pc pre-ISA) MT) 1 0))
        specultv?
        (b-nor modified? (if (MT-no-write-at (ISA-pc pre-ISA) MT) 1 0))
        0
        0
        '(IFU)
        0
        pre-ISA
        (ISA-step pre-ISA (ISA-input 0))))

(defun exintr-INST (MT pre-ISA modified?)
  (declare (xargs :guard (and (ISA-state-p pre-ISA) (MAETT-p MT))))
  (INST (MT-new-ID MT)
        modified?
        0
        0
        0
        1
        '(retire)
        0
        pre-ISA
        (ISA-step pre-ISA (ISA-input 1))))

; Coerce-DQ-stg coerces argument len to an integer between 0 and 3.
; The result is an index to a dispatch queue entry.
(defun coerce-DQ-stg (len)
  (declare (xargs :guard t))
  (if (and (integerp len) (> len 3)) 3 (nfix len)))

(defthm type-coerce-DQ-stg
    (and (integerp (coerce-DQ-stg len))
         (<= 0 (coerce-DQ-stg len)))
  :rule-classes :type-prescription)

(defthm range-coerce-DQ-stg
    (and (<= 0 (coerce-DQ-stg len))
         (<= (coerce-DQ-stg len) 3))
  :rule-classes :linear)
(in-theory (disable coerce-DQ-stg))

(defun new-DQ-stage (MT MA)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA))))
  (let ((len (MT-dq-len MT)))
    (list 'DQ (b-if (dispatch-inst? MA)
                    (coerce-DQ-stg (1- len))
                    (coerce-DQ-stg len)))))

(defthm dq-stg-p-new-dq-stg
    (DQ-stg-p (new-DQ-stage MT MA))
  :hints (("goal" :in-theory (enable new-dq-stage dq-stg-p))))

(defun step-INST-IFU (i MT MA sigs)
  (declare (xargs :guard (and (INST-p i)
                              (MAETT-p MT)
                              (MA-state-p MA)
                              (MA-input-p sigs))))
  (b-if (DQ-full? (MA-DQ MA))
        i
        (b-if (IFU-branch-predict? (MA-IFU MA) MA sigs)
              (update-INST i :stg (new-DQ-stage MT MA)
                           :br-predict? 1)
              (update-INST i :stg (new-DQ-stage MT MA)
```

```
                          :br-predict? 0))))

(defthm INST-p-step-INST-IFU
    (implies (and (INST-p i) (MAETT-p MT)
                  (MA-state-p MA) (MA-input-p sigs))
             (INST-p (step-INST-IFU i MT MA sigs))))
(in-theory (disable step-INST-IFU))

(defun dispatch-INST (i MA sigs)
  (declare (xargs :guard (and (INST-p i)
                              (MA-state-p MA) (MA-input-p sigs))))
  (cond ((b1p (dispatch-no-unit? MA))
         (update-INST i :stg '(complete)
                      :tag (ROB-tail (MA-ROB MA))))
        ((b1p (dispatch-to-IU? MA))
         (b-if (select-IU-RS0? (MA-IU MA))
               (update-INST i :stg '(IU RS0)
                            :tag (ROB-tail (MA-ROB MA)))
               ; must be (select-IU-RS1? (MA-IU MA))
               (update-INST i :stg '(IU RS1)
                            :tag (ROB-tail (MA-ROB MA)))))
        ((b1p (dispatch-to-MU? MA))
         (b-if (select-MU-RS0? (MA-MU MA))
               (update-INST i :stg '(MU RS0)
                            :tag (ROB-tail (MA-ROB MA)))
               ; must be (select-MU-RS1? (MA-MU MA))
               (update-INST i :stg '(MU RS1)
                            :tag (ROB-tail (MA-ROB MA)))))
        ((b1p (dispatch-to-BU? MA))
         (b-if (select-BU-RS0? (MA-BU MA))
               (update-INST i :stg '(BU RS0)
                            :tag (ROB-tail (MA-ROB MA)))
               ; must be (select-BU-RS1? (MA-BU MA))
               (update-INST i :stg '(BU RS1)
                            :tag (ROB-tail (MA-ROB MA)))))
        (t ; must be (dispatch-to-LSU? MA)
         (b-if (select-LSU-RS0? (MA-LSU MA))
               (update-INST i :stg '(LSU RS0)
                            :tag (ROB-tail (MA-ROB MA)))
               ; must be (select-LSU-RS1? (MA-BU MA))
               (update-INST i :stg '(LSU RS1)
                            :tag (ROB-tail (MA-ROB MA)))))))

(defun step-INST-DQ (i MT MA sigs)
  (declare (xargs :guard (and (INST-p i) (DQ-stg-p (INST-stg i))
                              (MAETT-p MT)
                              (MA-state-p MA) (MA-input-p sigs))))
  (let ((idx (DQ-stg-idx (INST-stg i))))
    (b-if (dispatch-inst? MA)
          (if (zp idx)
              (dispatch-INST i MA sigs)
              (update-INST i :stg (list 'DQ (nfix (1- idx)))))
          i)))

(defthm INST-p-step-INST-DQ
    (implies (and (INST-p i) (MAETT-p MT)
                  (MA-state-p MA) (MA-input-p sigs))
             (INST-p (step-INST-DQ i MT MA sigs))))
(in-theory (disable step-INST-DQ))

(defun step-INST-IU (i MA sigs)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA)
```

```
                              (IU-stg-p (INST-stg i))
                              (MA-input-p sigs))))
  (case (cadr (INST-stg i))
    (RS0 (b-if (issue-IU-RS0? (MA-IU MA) MA)
               (update-INST i :stg '(complete))
               i))
    (otherwise ; RS1
     (b-if (issue-IU-RS1? (MA-IU MA) MA)
           (update-INST i :stg '(complete))
           i))))

(defun step-INST-MU (i MA sigs)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA)
                              (MA-input-p sigs))))
  (let ((MU (MA-MU MA)))
    (case (cadr (INST-stg i))
      (RS0 (b-if (issue-MU-RS0? (MA-MU MA) MA)
                 (update-INST i :stg '(MU lch1))
                 i))
      (RS1
       (b-if (issue-MU-RS1? (MA-MU MA) MA)
             (update-INST i :stg '(MU lch1))
             i))
      (lch1
       (b-if (b-ior (CDB-for-MU? MA)
                    (b-not (MU-latch2-valid? (MU-lch2 MU))))
             (update-INST i :stg '(MU lch2))
             i))
      (otherwise ; lch2
       (b-if (CDB-for-MU? MA)
             (update-INST i :stg '(complete))
             i)))))

(defun step-INST-BU (i MA sigs)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA)
                              (MA-input-p sigs))))
  (let ((BU (MA-BU MA)))
    (case (cadr (INST-stg i))
      (RS0 (b-if (issue-BU-RS0? BU MA)
                 (update-INST i :stg '(complete))
                 i))
      (otherwise                          ; RS1
       (b-if (issue-BU-RS1? BU MA)
             (update-INST i :stg '(complete))
             i)))))

(defun INST-select-wbuf0? (MA sigs)
  (declare (xargs :guard (and (MA-state-p MA) (MA-input-p sigs))))
  (b-ior (b-not (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
         (b-and (b-not (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                (release-wbuf0? (MA-LSU MA) sigs))))

(defun step-INST-LSU-RS0 (i MA sigs)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA) (MA-input-p sigs))))
  (let ((RS0 (LSU-RS0 (MA-LSU MA)))
        (LSU (MA-LSU MA)))
    (b-if (issue-LSU-RS0? LSU MA sigs)
          (b-if (LSU-RS-ld-st? RS0)
                (b-if (INST-select-wbuf0? MA sigs)
                      (update-INST i :stg '(LSU wbuf0))
                ; if wbuf1 is not selected
                      (update-INST i :stg '(LSU wbuf1)))
```

```
              (update-INST i :stg '(LSU rbuf)))
         i)))

(defun step-INST-LSU-RS1 (i MA sigs)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA) (MA-input-p sigs))))
  (let ((RS1 (LSU-RS1 (MA-LSU MA)))
        (LSU (MA-LSU MA)))
    (b-if (issue-LSU-RS1? LSU MA sigs)
          (b-if (LSU-RS-ld-st? RS1)
                (b-if (INST-select-wbuf0? MA sigs)
                      (update-INST i :stg '(LSU wbuf0))
                ; if not, wbuf1 is selected
                      (update-INST i :stg '(LSU wbuf1)))
                (update-INST i :stg '(LSU rbuf)))
          i)))

(defun step-INST-LSU-rbuf (i MA sigs)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA) (MA-input-p sigs))))
  (let ((LSU (MA-LSU MA)))
    (b-if (release-rbuf? LSU MA sigs)
          (update-INST i :stg '(LSU lch))
          i)))

(defun step-INST-LSU-lch (i)
  (declare (xargs :guard (INST-p i)))
  (update-INST i :stg '(complete)))

(defun step-INST-LSU-wbuf0 (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (let ((LSU (MA-LSU MA)))
    (b-if (check-wbuf0? LSU)
          (update-INST i :stg '(LSU wbuf0 lch))
          i)))

(defun step-INST-LSU-wbuf0-lch (i)
  (declare (xargs :guard (INST-p i)))
  (update-INST i :stg '(complete wbuf0)))

(defun step-INST-LSU-wbuf1 (i MA sigs)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA) (MA-input-p sigs))))
  (let ((LSU (MA-LSU MA)))
    (b-if (check-wbuf1? LSU)
          (update-INST i :stg '(LSU wbuf1 lch))
          (b-if (release-wbuf0? LSU sigs)
                (update-INST i :stg '(LSU wbuf0))
                i))))

(defun step-INST-LSU-wbuf1-lch (i MA sigs)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA) (MA-input-p sigs))))
  (b-if (release-wbuf0? (MA-LSU MA) sigs)
        (update-INST i :stg '(complete wbuf0))
        (update-INST i :stg '(complete wbuf1))))

(defun step-INST-LSU (i MA sigs)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA) (MA-input-p sigs))))
  (cond ((equal (INST-stg i) '(LSU RS0)) (step-INST-LSU-RS0 i MA sigs))
        ((equal (INST-stg i) '(LSU RS1)) (step-INST-LSU-RS1 i MA sigs))
        ((equal (INST-stg i) '(LSU rbuf))
         (step-INST-LSU-RBUF i MA sigs))
        ((equal (INST-stg i) '(LSU lch)) (step-INST-LSU-lch i))
        ((equal (INST-stg i) '(LSU wbuf0))
         (step-INST-LSU-wbuf0 i MA))
```

```
        ((equal (INST-stg i) '(LSU wbuf0 lch))
         (step-INST-LSU-wbuf0-lch i))
        ((equal (INST-stg i) '(LSU wbuf1))
         (step-INST-LSU-wbuf1 i MA sigs))
        (t ; must be '(LSU wbuf1 lch)
         (step-INST-LSU-wbuf1-lch i MA sigs))))

(defun step-INST-execute (i MA sigs)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA) (MA-input-p sigs))))
  (let ((stg (INST-stg i)))
    (cond ((IU-stg-p stg) (step-INST-IU i MA sigs))
          ((MU-stg-p stg) (step-INST-MU i MA sigs))
          ((BU-stg-p stg) (step-INST-BU i MA sigs))
          (t ; must be (LSU-stg-p stg)
           (step-INST-LSU i MA sigs)))))

(defthm INST-p-step-INST-execute
    (implies (and (INST-p i) (MAETT-p MT)
                  (MA-state-p MA) (MA-input-p sigs))
             (INST-p (step-INST-execute i MA sigs))))
(in-theory (disable step-INST-execute))

; Describes the behavior of instruction i at the complete stage.  If
; it commits, an instruction generally retires. However, if it is a
; memory store instruction, we may still have a corresponding entry in
; the write buffer. In that case, the instruction goes into the commit
; stage, where the actual write operation takes place.  However, if
; exception is already detected for i, i retires when it is committed.
; So instructions at the commit stage cannot be an exception causing
; instruction.  An exception must be detected before the commit occurs
; in order to implement precise exceptions.
(defun step-INST-complete (i MA sigs)
  (declare (xargs :guard (and (INST-p i)
                              (MA-state-p MA) (MA-input-p sigs))))
  (cond ((equal (INST-stg i) '(complete))
         (b-if (INST-commit? i MA)
               (update-INST i :stg '(retire))
               i))
        ((equal (INST-stg i) '(complete wbuf0))
         (b-if (b-and (INST-commit? i MA) (enter-excpt? MA))
               (update-INST i :stg '(retire))
         (b-if (INST-commit? i MA)
               (update-INST i :stg '(commit wbuf0))
               i)))
        (t ; must be (equal (INST-stg i) '(complete wbuf1))
         (b-if (b-and (INST-commit? i MA) (enter-excpt? MA))
               (update-INST i :stg '(retire))
         (b-if (b-and (INST-commit? i MA) (release-wbuf0? (MA-LSU MA) sigs))
               (update-INST i :stg '(commit wbuf0))
         (b-if (INST-commit? i MA)
               (update-INST i :stg '(commit wbuf1))
         (b-if (release-wbuf0? (MA-LSU MA) sigs)
               (update-INST i :stg '(complete wbuf0))
               i)))))))

(defthm INST-p-step-INST-complete
    (implies (and (INST-p i) (MAETT-p MT)
                  (MA-state-p MA) (MA-input-p sigs))
             (INST-p (step-INST-complete i MA sigs))))
(in-theory (disable step-INST-complete))

(defun step-INST-commit (i MA sigs)
```

```
  (declare (xargs :guard (and (INST-p i)
                              (MA-state-p MA) (MA-input-p sigs))))
  (if (equal (INST-stg i) '(commit wbuf0))
      (if (b1p (release-wbuf0? (MA-LSU MA) sigs))
          (update-INST i :stg '(retire))
          i)
      ; must be (equal (INST-stg i) '(commit wbuf1))
      (if (b1p (release-wbuf0? (MA-LSU MA) sigs))
          (update-INST i :stg '(commit wbuf0))
          i)))

(defthm INST-p-step-INST-commit
    (implies (and (INST-p i) (MAETT-p MT)
                  (MA-state-p MA) (MA-input-p sigs))
             (INST-p (step-INST-commit i MA sigs))))
(in-theory (disable step-INST-commit))

(defun step-INST (i MT MA sigs)
  (declare (xargs :guard (and (INST-p i) (MAETT-p MT)
                              (MA-state-p MA) (MA-input-p sigs))))
  (cond ((IFU-stg-p (INST-stg i))
         (step-INST-IFU i MT MA sigs))
        ((DQ-stg-p (INST-stg i))
         (step-INST-DQ i MT MA sigs))
        ((execute-stg-p (INST-stg i))
         (step-INST-execute i MA sigs))
        ((complete-stg-p (INST-stg i))
         (step-INST-complete i MA sigs))
        ((commit-stg-p (INST-stg i))
         (step-INST-commit i MA sigs))
        (t ; retire
         i)))

(defthm INST-p-step-INST
    (implies (and (INST-p i) (MAETT-p MT)
                  (MA-state-p MA) (MA-input-p sigs))
             (INST-p (step-INST i MT MA sigs))))
(in-theory (disable step-INST))

(defun fetch-inst? (MA sigs)
  (declare (xargs :guard (and (MA-state-p MA) (MA-input-p sigs))
                  :guard-hints (("goal" :in-theory
                                        (enable IFU-fetch-prohibited?)))))
  (bs-and (b-not (flush-all? MA sigs))
          (b-not (IFU-branch-predict? (MA-IFU MA) MA sigs))
          (b-nand (IFU-valid? (MA-IFU MA)) (DQ-full? (MA-DQ MA)))
          (b-ior (MA-input-fetch sigs)
                 (IFU-fetch-prohibited? (MA-pc MA) (MA-mem MA)
                                        (SRF-su (MA-SRF MA))))))

(defun step-MT-dq-len (MT MA sigs)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))))
  (b-if (flush-all? MA sigs)
        0
        (b-if (dispatch-inst? MA)
              (b-if (b-andc1 (DQ-full? (MA-DQ MA)) (IFU-valid? (MA-IFU MA)))
                    (nfix (MT-dq-len MT))
                    (nfix (1- (MT-dq-len MT))))
              (b-if (b-andc1 (DQ-full? (MA-DQ MA)) (IFU-valid? (MA-IFU MA)))
                    (nfix (1+ (MT-dq-len MT)))
                    (nfix (MT-dq-len MT))))))
```

```
(defthm type-step-MT-dq-len
    (and (integerp (step-MT-dq-len MT MA sigs))
         (<= 0 (step-MT-dq-len MT MA sigs)))
  :rule-classes
  ((:type-prescription)
   (:rewrite :corollary
             (integerp (step-MT-dq-len MT MA sigs)))
   (:linear  :corollary
             (<= 0 (step-MT-dq-len MT MA sigs)))))

(in-theory (disable step-MT-dq-len))

(defun step-MT-wb-len (MT MA sigs)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))))
  (let ((LSU (MA-LSU MA)))
    (b-if (flush-all? MA sigs)
          (b-if (release-wbuf0? LSU sigs)
                (b-if (b-and (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))
                             (wbuf-commit? (LSU-wbuf1 (MA-LSU MA))))
                      1 0)
                (b-if (b-and (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))
                             (wbuf-commit? (LSU-wbuf1 (MA-LSU MA))))
                      2
                      (b-if (b-and (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))
                                   (wbuf-commit? (LSU-wbuf0 (MA-LSU MA))))
                            1 0)))
          (b-if (b-ior (b-and (issue-LSU-RS0? LSU MA sigs)
                              (LSU-RS-ld-st? (LSU-RS0 LSU)))
                       (b-and (issue-LSU-RS1? LSU MA sigs)
                              (LSU-RS-ld-st? (LSU-RS1 LSU))))
                (b-if (release-wbuf0? LSU sigs)
                      (MT-wb-len MT)
                      (1+ (MT-wb-len MT)))
                (b-if (release-wbuf0? LSU sigs)
                      (nfix (1- (MT-wb-len MT)))
                      (MT-wb-len MT))))))

(defthm type-step-MT-wb-len
    (implies (and (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (and (integerp (step-MT-wb-len MT MA sigs))
                  (<= 0 (step-MT-wb-len MT MA sigs))))
  :rule-classes
  ((:type-prescription)
   (:rewrite :corollary
             (implies (and (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
                      (integerp (step-MT-wb-len MT MA sigs))))
   (:linear  :corollary
             (implies (and (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
                      (<= 0 (step-MT-wb-len MT MA sigs))))))
(in-theory (disable step-MT-wb-len))

(defun step-MT-ROB-flg (MT MA sigs)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))))
  (b-if (flush-all? MA sigs) 0
        (b-xor (MT-ROB-flg MT)
               (b-xor (b-and (commit-inst? MA)
                             (logbit *rob-index-size* (+ 1 (MT-ROB-head MT))))
                      (b-and (dispatch-inst? MA)
                             (logbit *rob-index-size* (+ 1 (MT-ROB-tail MT))))))))

(defun step-MT-ROB-head (MT MA sigs)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))))
```

```
  (b-if (flush-all? MA sigs) 0
        (b-if (commit-inst? MA)
              (rob-index (+ 1 (MT-ROB-head MT)))
              (MT-ROB-head MT))))

(defthm rob-index-p-step-MT-rob-head
    (implies (and (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (rob-index-p (step-MT-rob-head MT MA sigs))))
(in-theory (disable step-MT-rob-head))

(defun step-MT-ROB-tail (MT MA sigs)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))))
  (b-if (flush-all? MA sigs) 0
        (b-if (dispatch-inst? MA)
              (rob-index (+ 1 (MT-ROB-tail MT)))
              (MT-ROB-tail MT))))

(defthm rob-index-p-step-MT-rob-tail
    (implies (and (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (rob-index-p (step-MT-rob-tail MT MA sigs))))
(in-theory (disable step-MT-rob-tail))


; MAETT trace updating function.
(defun step-trace (INST-list MT MA sigs pre-ISA specultv? modified?)
  (declare (xargs :guard (and (INST-listp INST-list)
                              (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                              (ISA-state-p pre-ISA) (bitp specultv?)
                              (bitp modified?))))
  (if (endp INST-list)
      (b-if (fetch-inst? MA sigs)
            (list (fetched-inst MT pre-ISA specultv? modified?))
            (b-if (ex-intr? MA sigs)
                  (list (exintr-INST MT pre-ISA modified?))
                  nil))
      (b-if (INST-cause-jmp? (car INST-list) MT MA sigs)
            (list (step-INST (car INST-list) MT MA sigs))
      (b-if (INST-exintr-now? (car INST-list) MA sigs)
            (list (exintr-INST MT pre-ISA modified?))
            (cons (step-INST (car INST-list) MT MA sigs)
                  (step-trace (cdr INST-list) MT MA sigs
                              (INST-post-ISA (car INST-list))
                              (b-ior (inst-specultv? (car INST-list))
                                     (INST-start-specultv? (car INST-list)))
                              (INST-modified? (car INST-list))))))))

(defthm INST-listp-step-MT-trace
    (implies (and (INST-listp INST-list)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (ISA-state-p ISA) (bitp spc) (bitp smc))
             (INST-listp (step-trace INST-list MT MA sigs ISA spc smc))))

; MAETT step function.
(defun MT-step (MT MA sigs)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))))
  (MAETT (MT-init-ISA MT)
         (1+ (MT-new-ID MT))
         (step-MT-dq-len MT MA sigs)
         (step-MT-wb-len MT MA sigs)
         (step-MT-rob-flg MT MA sigs)
         (step-MT-rob-head MT MA sigs)
         (step-MT-rob-tail MT MA sigs)
```

```
        (step-trace (MT-trace MT) MT MA sigs (MT-init-ISA MT) 0 0)))

(defthm MAETT-p-MT-step
    (implies (and (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (MAETT-p (MT-step MT MA sigs))))

(in-theory (disable MT-step))

; MAETT N-step function.
(defun MT-stepn (MT MA sig-list n)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA)
                              (MA-input-listp sig-list)
                              (integerp n)
                              (>= n 0))))
  (if (zp n)
      MT
      (if (endp sig-list)
          MT
          (MT-stepn (MT-step MT MA (car sig-list))
                    (MA-step MA (car sig-list))
                    (cdr sig-list)
                    (1- n)))))

(defthm MAETT-p-MT-stepn
    (implies (and (MAETT-p MT) (MA-state-p MA) (MA-input-listp sig-list))
             (MAETT-p (MT-stepn MT MA sig-list n))))

(deflabel end-of-MT-def)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Definition of init-MT
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun init-MT (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (MAETT (proj MA)
         0 0 0 0                          ; ID  DQ-len  WB-len ROB-flg
         (rob-head (MA-ROB MA))           ; rob-head
         (rob-tail (MA-ROB MA))           ; rob-tail
         nil))                            ; trace

(defthm MAETT-p-init-MT
    (implies (MA-state-p MA) (MAETT-p (init-MT MA))))
(in-theory (disable init-MT))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; An alternative definition of MT-step
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  We have an alternative definition of MT-step, which is defined as
;  MT-step*.  MT-step and MT-stepn* are slightly different.
;  Because I need a some functional definitions used in the
;  definition of MT-step.  See invariant-proof.lisp
;
;  Script Deleted in this file.


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; INST-at-stg
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deflabel begin-INST-at-functions)

(defun INST-at-stg-in-trace (stg trace)
  (declare (xargs :guard (INST-listp trace)))
```

```
  (if (endp trace)
      nil
      (if (equal stg (INST-stg (car trace)))
          (car trace)
          (INST-at-stg-in-trace stg (cdr trace)))))

(defun INST-at-stg (stg MT)
  (declare (xargs :guard (and (stage-p stg) (MAETT-p MT))))
  (INST-at-stg-in-trace stg (MT-trace MT)))

(defun no-INST-at-stg-in-trace (stg trace)
  (declare (xargs :guard (and (stage-p stg) (INST-listp trace))))
  (if (endp trace)
      T
      (if (equal (INST-stg (car trace)) stg)
          nil
          (no-INST-at-stg-in-trace stg (cdr trace)))))

(defun uniq-INST-at-stg-in-trace (stg trace)
  (declare (xargs :guard (and (stage-p stg) (INST-listp trace))))
  (if (endp trace)
      nil
      (if (equal (INST-stg (car trace)) stg)
          (no-INST-at-stg-in-trace stg (cdr trace))
          (uniq-INST-at-stg-in-trace stg (cdr trace)))))

(defun no-INST-at-stg (stg MT)
  (declare (xargs :guard (and (stage-p stg) (MAETT-p MT))))
  (no-INST-at-stg-in-trace stg (MT-trace MT)))

(defun uniq-INST-at-stg (stg MT)
  (declare (xargs :guard (and (stage-p stg) (MAETT-p MT))))
  (uniq-INST-at-stg-in-trace stg (MT-trace MT)))

(defun INST-at-stgs-in-trace (stgs trace)
  (declare (xargs :guard (and (true-listp stgs) (INST-listp trace))))
  (if (endp trace)
      nil
      (if (member-equal (INST-stg (car trace)) stgs)
          (car trace)
          (INST-at-stgs-in-trace stgs (cdr trace)))))

(defun INST-at-stgs (stgs MT)
  (declare (xargs :guard (and (true-listp stgs) (MAETT-p MT))))
  (INST-at-stgs-in-trace stgs (MT-trace MT)))

(defun no-INST-at-stgs-in-trace (stgs trace)
  (declare (xargs :guard (and (true-listp stgs) (INST-listp trace))))
  (if (endp trace)
      T
      (if (member-equal (INST-stg (car trace)) stgs)
          nil
          (no-INST-at-stgs-in-trace stgs (cdr trace)))))

(defun uniq-INST-at-stgs-in-trace (stgs trace)
  (declare (xargs :guard (and (true-listp stgs) (INST-listp trace))))
  (if (endp trace)
      nil
      (if (member-equal (INST-stg (car trace)) stgs)
          (no-INST-at-stgs-in-trace stgs (cdr trace))
          (uniq-INST-at-stgs-in-trace stgs (cdr trace)))))
```

```
(defun no-INST-at-stgs (stgs MT)
  (declare (xargs :guard (and (true-listp stgs) (MAETT-p MT))))
  (no-INST-at-stgs-in-trace stgs (MT-trace MT)))

(defun uniq-INST-at-stgs (stgs MT)
  (declare (xargs :guard (and (true-listp stgs) (MAETT-p MT))))
  (uniq-INST-at-stgs-in-trace stgs (MT-trace MT)))

(defun inst-of-tag-in-trace (tg trace)
  (declare (xargs :guard (and (rob-index-p tg) (INST-listp trace))))
  (if (endp trace)
      nil
      (if (and (equal (inst-tag (car trace)) tg)
               (dispatched-p (car trace))
               (not (committed-p (car trace))))
          (car trace)
          (inst-of-tag-in-trace tg (cdr trace)))))

(defun inst-of-tag (tg MT)
  (declare (xargs :guard (and (rob-index-p tg) (MAETT-p MT))))
  (inst-of-tag-in-trace tg (MT-trace MT)))

(defun no-inst-of-tag-in-trace (tg trace)
  (declare (xargs :guard (and (rob-index-p tg) (INST-listp trace))))
  (if (endp trace)
      T
      (if (and (equal (inst-tag (car trace)) tg)
               (dispatched-p (car trace))
               (not (committed-p (car trace))))
          nil
          (no-inst-of-tag-in-trace tg (cdr trace)))))

(defun uniq-inst-of-tag-in-trace (tg trace)
  (declare (xargs :guard (and (rob-index-p tg) (INST-listp trace))))
  (if (endp trace)
      nil
      (if (and (equal (inst-tag (car trace)) tg)
               (dispatched-p (car trace))
               (not (committed-p (car trace))))
          (no-inst-of-tag-in-trace tg (cdr trace))
          (uniq-inst-of-tag-in-trace tg (cdr trace)))))

(defun no-inst-of-tag (tg MT)
  (declare (xargs :guard (and (rob-index-p tg) (MAETT-p MT))))
  (no-inst-of-tag-in-trace tg (MT-trace MT)))

(defun uniq-inst-of-tag (tg MT)
  (declare (xargs :guard (and (rob-index-p tg) (MAETT-p MT))))
  (uniq-inst-of-tag-in-trace tg (MT-trace MT)))

(deflabel end-INST-at-functions)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Defining Theories
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deftheory MA-stg-def
    (set-difference-theories (function-theory 'end-MA-stg-def)
                             (function-theory 'begin-MA-stg-def)))

(deftheory MT-def
    (set-difference-theories (function-theory 'end-of-MT-def)
                             (function-theory 'begin-of-MT-def)))
```

```
(deftheory MT-def-non-rec-functions
    (non-rec-functions (theory 'MT-def) world))

(deftheory INST-at-functions
    (set-difference-theories (function-theory 'end-INST-at-functions)
                             (function-theory 'begin-INST-at-functions)))

(deftheory INST-function-def
    (definition-theory
        (set-difference-theories (universal-theory 'end-INST-function-def)
                                 (universal-theory 'begin-INST-function-def))))

(deftheory INST-at-non-rec-functions
    (non-rec-functions (theory 'INST-at-functions) world))

(in-theory (disable MT-def-non-rec-functions))
(in-theory (disable MA-stg-def))
(in-theory (disable INST-at-non-rec-functions))
(in-theory (disable INST-function-def))

(in-theory (enable INST-su INST-pc))

(deftheory step-INST-low-level-functions
    '(step-inst-IFU step-inst-DQ step-inst-execute
      step-INST-IU step-INST-MU step-INST-BU step-INST-LSU
      step-INST-LSU-RS0 step-INST-LSU-RS1 step-INST-LSU-rbuf
      step-INST-LSU-lch step-INST-LSU-wbuf0 step-INST-LSU-wbuf0-lch
      step-INST-LSU-wbuf1 step-INST-LSU-wbuf1-lch
      step-inst-complete
      step-inst-commit dispatch-inst))

#|
;; Eval-set-print-MA evaluates expression expr, set variable s to the result,
;; and print out the result briefly.
;;
;; Example (eval-set-print-MA s1 (MA-step (@ s) (MA-input 0 0 1 1)))
(defmacro eval-set-print-MA (s expr)
  `(pprogn (f-put-global ',s ,expr state)
           (mv nil
             (list (MA-pc (f-get-global ',s state))
                   (MA-RF (f-get-global ',s state))
                   (MA-SRF (f-get-global ',s state))
                   (MA-DQ (f-get-global ',s state))
                   (MA-ROB (f-get-global ',s state))
                   (MA-IU (f-get-global ',s state))
                   (MA-MU (f-get-global ',s state))
                   (MA-BU (f-get-global ',s state))
                   (MA-LSU (f-get-global ',s state)))
             state)))

(defmacro natp (n) `(and (integerp ,n) (<= 0 ,n)))

; (make-pair-seq from end) generates a list of pair of symbols
; beginning with s and MT.  For instance, (make-pair-seq 0 3)
; generates ((S0 MT0) (S1 MT1) (S2 MT2) (S3 MT3)).
(defun make-pair-seq (from end)
  (declare (xargs :mode :program))
  (if (or (not (natp from)) (not (natp end)) (zp (- end from)))
      nil
      (cons
       (list (pack-intern 's (coerce (append (coerce "S" 'list)
```

```
                                          (explode-nonnegative-integer from
                                                                       nil))
                                   'string))
            (pack-intern 's (coerce (append (coerce "MT" 'list)
                                          (explode-nonnegative-integer from
                                                                       nil))
                                   'string)))
       (make-pair-seq (1+ from) end))))

;; A help function.
(defun make-MA-step-seq (sigs seq)
  (if (endp seq) nil
      (if (endp (cdr seq)) nil
          (let ((new-s (car (cadr seq)))
                (new-m (cadr (cadr seq)))
                (old-s (car (car seq)))
                (old-m (cadr (car seq))))
            (cons `(f-put-global ',new-s (MA-step (@ ,old-s) ,sigs)
                                 state)
                  (cons `(f-put-global ',new-m (MT-step (@ ,old-m)
                                                        (@ ,old-s)
                                                        ,sigs)
                                       state)
                        (make-MA-step-seq sigs (cdr seq))))))))

; Apply MA-step repeatedly and bind the generated MA states to the variables
; given in list seq.  Sigs is supplied to MA-step at every transition.
; Example:
; (MT-step-seq (MA-input 0 0 1 1) ((S0 MT0) (S1 MT1) (S2 MT2) (S3 MT3)))
(defmacro MT-step-seq (sigs seq)
  (if (endp seq) nil
      (let ((last-s (car (car (last seq)))))
        `(pprogn
          ,@(make-MA-step-seq sigs seq)
          (mv nil
              (list (MA-pc (f-get-global ',last-s state))
                    (MA-RF (f-get-global ',last-s state))
                    (MA-SRF (f-get-global ',last-s state))
                    (MA-ROB (f-get-global ',last-s state)))
              state)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Printing functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun pr-ISA (s)
  (list (ISA-pc s) (ISA-RF s) (ISA-SRF s)))

(defun pr-INST (i)
  (list (INST-ID i) (ISA-pc (INST-pre-ISA i))
        (inst-specultv? i) (INST-word i) (INST-stg i)
        (inst-tag i)))

(defun pr-INST-list (mlist)
  (if (endp mlist)
      nil
      (cons (pr-INST (car mlist)) (pr-INST-list (cdr mlist)))))

; Typical Usage:
; (pr-MT mt0)
(defmacro pr-MT (MT)
  `(list (pr-isa (MT-init-ISA (@ ,MT)))
         (MT-new-ID (@ ,MT))
```

```
         (MT-dq-len (@ ,MT))
         (MT-wb-len (@ ,MT))
         (MT-rob-head (@ ,MT))
         (MT-rob-tail (@ ,MT))
         (pr-INST-list (MT-trace (@ ,MT)))))

; Typical Usage:
; (pr-MA s0)
(defmacro pr-MA (s)
  `(list (MA-pc (@ ,s)) (MA-RF (@ ,s)) (MA-SRF (@ ,s))
    (MA-IFU (@ ,s)) (MA-DQ (@ ,s)) (MA-ROB (@ ,s)) (MA-IU (@ ,s))
    (MA-MU (@ ,s)) (MA-BU (@ ,s)) (MA-LSU (@ ,s))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Initial State Setting
(progn
(assign RF '(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
(assign SRF (SRF 1 0 0))

(assign IFU (IFU 0 0 0 0))
(assign DE (dispatch-entry 0 0 0 0 0 0 0 0))
(assign reg-s (reg-ref 0 0))
(assign reg-tbl (make-list *num-regs* :initial-element (@ reg-s)))
(assign sreg-tbl (sreg-tbl (@ reg-s) (@ reg-s)))
(assign DQ (DQ (@ DE) (@ DE) (@ DE) (@ DE) (@ reg-tbl) (@ sreg-tbl)))

(assign ROBE (ROB-entry 0 0 0 0 0 0 0 0 0 0 0 0 0))
(assign entries (make-list *rob-size* :initial-element (@ ROBE)))
(assign ROB (ROB 0 0 0 0 (@ entries)))

(assign IU (integer-unit (RS 0 0 0 0 0 0 0 0 0) (RS 0 0 0 0 0 0 0 0 0)))
(assign MU (mult-unit (RS 0 0 0 0 0 0 0 0 0) (RS 0 0 0 0 0 0 0 0 0)
                      (MU-latch1 0 0 0) (MU-latch2 0 0 0)))
(assign BU (branch-unit (BU-RS 0 0 0 0 0) (BU-RS 0 0 0 0 0)))
(assign LSU (load-store-unit 0
                             (LSU-RS 0 0 0 0  0 0 0 0 0 0 0 0 0)
                             (LSU-RS 0 0 0 0  0 0 0 0 0 0 0 0 0)
                             (read-buffer 0 0 0 0 0)
                             (write-buffer 0 0 0 0 0 0)
                             (write-buffer 0 0 0 0 0 0)
                             (LSU-latch 0 0 0 0)))
(assign mem-alist '(
; Exception Handler
(#x0 . #x7050) ; ST R0,(#x50)
(#x1 . #x6003) ; LD R0,(#x3)
(#x2 . #x2000) ; BZ R0, 0
(#x3 . 0)
; Exception Handler
(#x10 . #x7050) ; ST R0,(#x50)
(#x11 . #x6013) ; LD R0,(#x13)
(#x12 . #x2000) ; BZ R0, 0
(#x13 . 0)
; Exception Handler
(#x20 . #x7050) ; ST R0,(#x50)
(#x21 . #x6023) ; LD R0,(#x23)
(#x22 . #x2000) ; BZ R0, 0
(#x23 . 0)

; Exception Handler
(#x30 . #x7050) ; ST R0,(#x50)
(#x31 . #x6033) ; LD R0,(#x33)
(#x32 . #x2000) ; BZ R0, 0
```

```
(#x33 . 0)

; Kernel Data Section
(#x60 .  0)
(#x61 . 1)
(#x62 . 2)
(#x63 . #xFFFF) ; -1
(#x70 . #x400)
(#x71 . #x800)
; Kernel Dispatching code
(#x100 . #x6F70) ; LD R15,(#x70) ; program base
(#x101 . #x6E71) ;  LD R14,(#x71) ; data base
(#x102 . #x6060) ;  LD R0, (#x60) ; 0
(#x103 . #x6161) ;  LD R1, (#x61) ; 1
(#x104 . #x6262) ; LD R2, (#x62) ; 2
(#x105 . #x6363) ; LD R3, (#x63) ; -1
(#x106 . #xAF00) ;  MTSR SR0,R15
(#x107 . #xA010) ;  MTSR SR1,R0
(#x108 . #x8000) ; #x103: RFEH
; Program
(#x400 . #x35E0) ;  LD R5,(R14+R0) ; R5 holds counter
(#x401 . #x0601) ;  ADD R6, R0, R1     ; R6 holds factorial. Initially 1.
; Loop:
(#x402 . #x1665) ;  Mul R6, R6, R5  ; counter * fact -> fact
(#x403 . #x0553) ;  ADD R5, R5, R3  ; decrement fact
(#x404 . #x2502) ;  BZ  R5, Exit; if counter is zero, exit
(#x405 . #x20FD) ;  BZ  R0, Loop ; always jump to loop
; EXIT:
(#x406 . #x46E1) ; ST  R6, (R14+R1)
(#x407 . #x5000) ; SYNC
(#x408 . #xB000) ; Trap

; Data Section
(#x800 . 5)
(#x801 . 0)
(#x802 . 5) ; Offset to Loop
(#x803 . 9) ; Offset to Exit
))

(assign mem (set-page-mode *read-only* 1 (compress1 'mem *init-mem*)))
(assign mem (set-page-mode *read-write* 2 (@ mem)))
(assign mem (compress1 'mem (load-mem-alist (@ mem-alist) (@ mem))))
(assign s0 (MA-state #x100 (@ RF) (@ SRF) (@ IFU) (@ DQ) (@ ROB)
                   (@ IU) (@ MU) (@ BU) (@ LSU) (@ mem)))

(assign MT0
        (MAETT (proj (@ s0))
               0 0 0 0 0 0 nil))

(assign sigs (MA-input 0 0 1 1))
)

|#
```

D.4 Invariant Definitions

The definition of the invariant condition inv, which is a conjunction of a number of properties about the FM9801 microarchitecture and its MAETT. Please see Chapter 8 for detailed discussions.

D.4.1 invariants-def.lisp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; invariants.lisp:
; Author  Jun Sawada, University of Texas at Austin
;
;  The definition of the invariant condition of the FM9801.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(in-package "ACL2")

(include-book "MAETT-def")

; Starting the definition of various invariants
(deflabel begin-invariants-def)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Definition of weak-inv.
;  Weak invariants are well-formedness predicate.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun trace-correct-modified-first (trace)
  (declare (xargs :guard  (INST-listp trace)))
  (if (endp trace) t
      (if (endp (cdr trace)) t
          (and (implies (and (b1p (INST-modified? (cadr trace)))
                             (not (b1p (INST-modified? (car trace)))))
                        (b1p (INST-first-modified? (cadr trace))))
               (trace-correct-modified-first (cdr trace))))))

(defun correct-modified-first (MT)
  (declare (xargs :guard  (MAETT-p MT)))
  (trace-correct-modified-first (MT-trace MT)))

(defun ID-lt-all-p (trace ID)
  (declare (xargs :guard (and (INST-listp trace) (integerp ID) (<= 0 ID))))
  (if (endp trace)
      T
      (and (< (INST-ID (car trace)) ID)
           (ID-lt-all-p (cdr trace) ID))))

(defun MT-new-ID-distinct-p (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (ID-lt-all-p (MT-trace MT) (MT-new-ID MT)))

(defun member-eq-ID (i trace)
  (declare (xargs :guard (and (INST-p i) (INST-listp trace))))
  (if (endp trace)
      nil
      (if (equal (INST-ID i) (INST-ID (car trace)))
          trace
          (member-eq-ID i (cdr trace)))))
```

```
; All ID of INST's are different.
(defun distinct-IDs-p (trace)
  (declare (xargs :guard (and (INST-listp trace))))
  (if (endp trace)
      T
      (and (not (member-eq-ID (car trace) (cdr trace)))
           (distinct-IDs-p (cdr trace)))))

(defun MT-distinct-IDs-p (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (distinct-IDs-p (MT-trace MT)))

(defun distinct-member-p (trace)
  (declare (xargs :guard (and (INST-listp trace))))
  (if (endp trace)
      T
      (and (not (member-equal (car trace) (cdr trace)))
           (distinct-member-p (cdr trace)))))

; No INST entry of MT equals to other INST's.
(defun MT-distinct-INST-p (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (distinct-member-p (MT-trace MT)))

(defun ISA-chained-trace-p (INST-list ISA)
  (declare (xargs :guard (and (INST-listp INST-list) (ISA-state-p ISA))))
  (if (endp INST-list)
      T
      (and (equal ISA (INST-pre-ISA (car INST-list)))
           (equal (ISA-step (INST-pre-ISA (car INST-list))
                            (ISA-input (INST-exintr? (car INST-list))))
                  (INST-post-ISA (car INST-list)))
           (ISA-chained-trace-p (cdr INST-list)
                                (INST-post-ISA (car INST-list))))))

; Pre-ISA and post-ISA states form a chain.
(defun ISA-step-chain-p (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (ISA-chained-trace-p (MT-trace MT) (MT-init-ISA MT)))

(defun trace-all-modified-p (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace) T
      (and (b1p (INST-modified? (car trace)))
           (trace-all-modified-p (cdr trace)))))

(defun trace-modified-flg-sticky-p (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace) t
      (if (b1p (INST-modified? (car trace)))
          (trace-all-modified-p (cdr trace))
          (trace-modified-flg-sticky-p (cdr trace)))))

(defun INST-modify-p (i j)
  (declare (xargs :guard (and (INST-p i) (INST-p j))))
  (and (b1p (INST-store? i))
       (equal (INST-store-addr i)
              (ISA-pc (INST-pre-ISA j)))
       (not (b1p (INST-excpt? i)))
       (not (b1p (INST-exintr? i)))
       (not (b1p (INST-exintr? j)))))
```

```
(defun trace-modify-p (i trace)
  (declare (xargs :guard (and (INST-p i) (INST-listp trace))))
  (if (endp trace) nil
      (if (equal (car trace) i)
          nil
          (or (INST-modify-p (car trace) i)
              (trace-modify-p i (cdr trace))))))

(defun MT-modify-p (i MT)
  (declare (xargs :guard (and (INST-p i) (MAETT-p MT))))
  (trace-modify-p i (MT-trace MT)))

(defun trace-correct-modified-flgs-p (trace MT sticky)
  (declare (xargs :guard (and (INST-listp trace) (MAETT-p MT) (bitp sticky))))
  (if (endp trace)
      T
      (if (MT-modify-p (car trace) MT)
          (and (equal (INST-modified? (car trace)) 1)
               (trace-correct-modified-flgs-p (cdr trace) MT 1))
          (and (equal (INST-modified? (car trace)) sticky)
               (trace-correct-modified-flgs-p (cdr trace) MT sticky)))))

; Modified? bit of each INST is sticky.
(defun correct-modified-flgs-p (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (trace-correct-modified-flgs-p (MT-trace MT) MT 0))

; Definition of weak invariants. Most of the invariants are
; about the well-formedness of the MAETT.
(defun weak-inv (MT)
  (declare (xargs :guard  (MAETT-p MT)))
  (and (MT-new-ID-distinct-p MT)
       (MT-distinct-IDs-p MT)
       (MT-distinct-INST-p MT)
       (ISA-step-chain-p MT)
       (correct-modified-flgs-p MT)
       (correct-modified-first MT)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Definition of Strong Invariants (or what we usually call invariants).
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun trace-final-ISA (trace pre)
  (declare (xargs :guard (and (INST-listp trace) (ISA-state-p pre))))
  (if (endp trace)
      pre
      (trace-final-ISA (cdr trace) (INST-post-ISA (car trace)))))

(defthm ISA-state-p-trace-final-ISA
    (implies (and (INST-listp trace) (ISA-state-p pre))
             (ISA-state-p (trace-final-ISA trace pre))))

(defun MT-final-ISA (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (trace-final-ISA (MT-trace MT) (MT-init-ISA MT)))

; MT-all-retired-p is a predicate to check whether all instructions in
; a MAETT are retired.
(defun trace-all-retired (INST-list)
  (declare (xargs :guard (INST-listp INST-list)))
  (if (endp INST-list)
```

```
      T
      (and (retire-stg-p (INST-stg (car INST-list)))
           (trace-all-retired (cdr INST-list)))))

(defun MT-all-retired-p (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (trace-all-retired (MT-trace MT)))

(in-theory (disable MT-all-retired-p))

(defthm ISA-state-p-MT-final-ISA
    (implies (MAETT-p MT) (ISA-state-p (MT-final-ISA MT)))
  :hints (("Goal" :in-theory (enable MT-final-ISA))))

(defun trace-self-modify? (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace)
      0
      (b-if (INST-modified? (car trace))
            1
            (trace-self-modify? (cdr trace)))))

(defthm bitp-trace-self-modify?
    (bitp (trace-self-modify? trace)))

(defun MT-self-modify? (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (trace-self-modify? (MT-trace MT)))

(defthm bitp-MT-self-modify?
    (bitp (MT-self-modify? MT)))

(defun MT-self-modify-p (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (b1p (MT-self-modify? MT)))

; MT-CMI-p checks whether MT contains a committed self-modified
; instruction.  The recursive function trace-CMI-p only checks the
; continuous segment of instructions in MAETT, which are all
; committed.  Since instructions commit in-order, we don't have to
; look further once we encounter a non-committed instruction.
(defun trace-CMI-p (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace)
      nil
      (or  (and (committed-p (car trace))
                (b1p (INST-modified? (car trace))))
           (and (committed-p (car trace))
                (trace-CMI-p (cdr trace))))))

(defun MT-CMI-p (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (trace-CMI-p (MT-trace MT)))

(defun trace-specultv? (INST-list)
  (declare (xargs :guard (INST-listp INST-list)))
  (if (endp INST-list)
      0
      (b-if (b-ior (inst-specultv? (car INST-list))
                   (INST-start-specultv? (car INST-list)))
            1
            (trace-specultv? (cdr INST-list)))))
```

```
(defthm bitp-trace-specultv?
    (bitp (trace-specultv? trace)))

(defun MT-specultv? (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (trace-specultv? (MT-trace MT)))

(defthm bitp-MT-specultv?
    (bitp (MT-specultv? MT))
  :hints (("Goal" :in-theory (enable MT-specultv?))))

(defun MT-specultv-p (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (b1p (MT-specultv? MT)))

(defun trace-specultv-at-dispatch? (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace)
      0
      (if (not (dispatched-p (car trace)))
          0
          (if (committed-p (car trace))
              (trace-specultv-at-dispatch? (cdr trace))
              (b-if (b-ior (inst-specultv? (car trace))
                           (INST-start-specultv? (car trace)))
                    1
                    (trace-specultv-at-dispatch? (cdr trace)))))))

(defthm bitp-trace-specultv-at-dispatch?
    (bitp (trace-specultv-at-dispatch? trace)))

(defun MT-specultv-at-dispatch? (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (trace-specultv-at-dispatch? (MT-trace MT)))

(defthm bitp-MT-specultv-at-dispatch?
    (bitp (MT-specultv-at-dispatch? MT)))

(defun trace-modified-at-dispatch? (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace)
      0
      (if (not (dispatched-p (car trace)))
          0
          (b-if (INST-modified? (car trace))
                1
                (trace-modified-at-dispatch? (cdr trace))))))

(defthm bitp-trace-modified-at-dispatch?
    (bitp (trace-modified-at-dispatch? trace)))

(defun MT-modified-at-dispatch? (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (trace-modified-at-dispatch? (MT-trace MT)))

(defthm bitp-MT-modified-at-dispatch?
    (bitp (MT-modified-at-dispatch? MT)))

(defun trace-pc (INST-list pre-pc)
  (declare (xargs :guard (and (INST-listp INST-list) (addr-p pre-pc))))
  (if (endp INST-list)
```

```
      pre-pc
      (trace-pc (cdr INST-list) (ISA-pc (INST-post-ISA (car INST-list))))))

(defthm addr-p-trace-pc
    (implies (and (addr-p pc) (INST-listp trace))
             (addr-p (trace-pc trace pc))))

(defun MT-pc (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (trace-pc (MT-trace MT) (ISA-pc (MT-init-ISA MT))))

(defthm addr-p-MT-pc
    (implies (MAETT-p MT) (addr-p (MT-pc MT)))
  :hints (("Goal" :in-theory (enable MT-pc))))

; The pc in MA is in the correct state.
(defun pc-match-p (MT MA)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA))))
  (implies (not (or (MT-specultv-p MT)
                    (MT-self-modify-p MT)))
           (equal (MT-pc MT) (MA-pc MA))))

(defun trace-RF (INST-list RF)
  (declare (xargs :guard (and (INST-listp INST-list) (RF-p RF))))
  (if (endp INST-list)
      RF
      (if (not (or (retire-stg-p (INST-stg (car INST-list)))
                   (commit-stg-p (INST-stg (car INST-list)))))
          RF
          (trace-RF (cdr INST-list)
                    (ISA-RF (INST-post-ISA (car INST-list)))))))

(defthm trace-RF*
    (equal (trace-RF INST-list RF)
           (if (endp INST-list)
               RF
               (if (not (committed-p (car INST-list)))
                   RF
                   (trace-RF (cdr INST-list)
                             (ISA-RF (INST-post-ISA (car INST-list))))))))
  :rule-classes (:definition))

(defthm RF-p-trace-RF
    (implies (and (INST-listp trace) (RF-p RF))
             (RF-p (trace-RF trace RF))))

(defun MT-RF (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (trace-RF (MT-trace MT) (ISA-RF (MT-init-ISA MT))))

(defthm RF-p-MT-RF
    (implies (MAETT-p MT) (RF-p (MT-RF MT))))

; The register file in MA is in the correct state.
(defun RF-match-p (MT MA)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA))))
  (equal (MT-RF MT) (MA-RF MA)))

(defun trace-SRF (INST-list SRF)
  (declare (xargs :guard (and (INST-listp INST-list) (SRF-p SRF))))
  (if (endp INST-list)
      SRF
```

```
      (if (not (or (retire-stg-p (INST-stg (car INST-list)))
                   (commit-stg-p (INST-stg (car INST-list)))))
          SRF
          (trace-SRF (cdr INST-list)
                     (ISA-SRF (INST-post-ISA (car INST-list))))))

(defthm SRF-p-trace-SRF
    (implies (and (SRF-p SRF) (INST-listp trace))
             (SRF-p (trace-SRF trace SRF))))

(defun MT-SRF (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (trace-SRF (MT-trace MT) (ISA-SRF (MT-init-ISA MT))))

(defthm SRF-p-MT-SRF
    (implies (MAETT-p MT) (SRF-p (MT-SRF MT))))

; The SRF in MA is in the correct state.
(defun SRF-match-p (MT MA)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA))))
  (equal (MT-SRF MT) (MA-SRF MA)))

(defun trace-mem (INST-list mem)
  (declare (xargs :guard (and (INST-listp INST-list) (mem-p mem))))
  (if (endp INST-list)
      mem
      (if (not (retire-stg-p (INST-stg (car INST-list))))
          mem
          (trace-mem (cdr INST-list)
                     (ISA-mem (INST-post-ISA (car INST-list)))))))

(defthm mem-p-trace-mem
    (implies (and (mem-p mem) (INST-listp trace))
             (mem-p (trace-mem trace mem))))

(defun MT-mem (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (trace-mem (MT-trace MT) (ISA-mem (MT-init-ISA MT))))

(defthm mem-p-MT-mem
    (implies (MAETT-p MT) (mem-p (MT-mem MT))))

; The memory in MA is in the correct state.
(defun mem-match-p (MT MA)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA))))
  (equal (MT-mem MT) (MA-mem MA)))

(defun trace-no-specultv-commit-p (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace)
      T
      (and (implies (committed-p (car trace))
                    (zbp (inst-specultv? (car trace))))
           (trace-no-specultv-commit-p (cdr trace)))))

; No speculatively executed instructions are committed.
(defun no-specultv-commit-p (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (trace-no-specultv-commit-p (MT-trace MT)))

(defun trace-all-specultv-p (trace)
  (declare (xargs :guard (INST-listp trace)))
```

```
  (if (endp trace)
      t
      (and (b1p (inst-specultv? (car trace)))
           (trace-all-specultv-p (cdr trace)))))

(defun trace-correct-speculation-p (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace) T
      (and (not (b1p (inst-specultv? (car trace))))
           (if (b1p (INST-start-specultv? (car trace)))
               (trace-all-specultv-p (cdr trace))
               (trace-correct-speculation-p (cdr trace))))))


; Speculatively executed instructions follows an instruction that
; satisfies INST-start-specultv?
(defun correct-speculation-p (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (trace-correct-speculation-p (MT-trace MT)))

(defun trace-correct-exintr-p (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace) t
      (and (implies (b1p (INST-exintr? (car trace)))
                    (retire-stg-p (INST-stg (car trace))))
           (trace-correct-exintr-p (cdr trace)))))

; External interrupt field is set only for a retired instruction.
(defun correct-exintr-p (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (trace-correct-exintr-p (MT-trace MT)))

; From here we define MT-INST-inv.
(deflabel begin-inst-inv-def)

; The condition that instruction i should satisfy at stage IFU.
; Following predicates are for other stages.
(defun IFU-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (IFU-valid? (MA-IFU MA)))
       ; This condition is added later.
       ; If this instruction is modified by a previous instruction,
       ; but no preceding instruction is neither modified or speculatively
       ; executed, at least the program counter value is correct.
       (implies (and (not (b1p (inst-specultv? i)))
                     (implies (b1p (INST-modified? i))
                              (b1p (INST-first-modified? i))))
                (equal (IFU-pc (MA-IFU MA)) (ISA-pc (INST-pre-ISA i))))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
        (and (equal (IFU-pc (MA-IFU MA)) (ISA-pc (INST-pre-ISA i)))
             (equal (IFU-excpt (MA-IFU MA)) (INST-excpt-flags i))
             (equal (IFU-word (MA-IFU MA))
                    (if (INST-fetch-error-detected-p i) 0 (INST-word i)))))))

(defun DQ0-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))
       (implies (and (not (b1p (inst-specultv? i)))
                     (implies (b1p (INST-modified? i))
                              (b1p (INST-first-modified? i))))
                (equal (DE-pc (DQ-DE0 (MA-DQ MA))) (ISA-pc (INST-pre-ISA i))))
```

```
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
        (and (equal (DE-excpt (DQ-DE0 (MA-DQ MA))) (INST-excpt-flags i))
             (equal (DE-pc (DQ-DE0 (MA-DQ MA))) (ISA-pc (INST-pre-ISA i)))
             (implies (not (INST-fetch-error-detected-p i))
              (and (equal (DE-cntlv (DQ-DE0 (MA-DQ MA))) (INST-cntlv i))
                   (equal (DE-rc (DQ-DE0 (MA-DQ MA)))
                          (rc (INST-word i)))
                   (equal (DE-ra (DQ-DE0 (MA-DQ MA)))
                          (ra (INST-word i)))
                   (equal (DE-rb (DQ-DE0 (MA-DQ MA)))
                          (rb (INST-word i)))
                   (equal (DE-im (DQ-DE0 (MA-DQ MA)))
                          (im (INST-word i)))
                   (equal (DE-br-target (DQ-DE0 (MA-DQ MA)))
                          (INST-br-target i))))
             (implies (INST-fetch-error-detected-p i)
                      (equal (DE-cntlv (DQ-DE0 (MA-DQ MA)))
                             (decode 0 (INST-br-predict? i))))))))

(defun DQ1-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))
       (implies (and (not (b1p (inst-specultv? i)))
                     (implies (b1p (INST-modified? i))
                              (b1p (INST-first-modified? i))))
                (equal (DE-pc (DQ-DE1 (MA-DQ MA))) (ISA-pc (INST-pre-ISA i))))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
        (and (equal (DE-excpt (DQ-DE1 (MA-DQ MA))) (INST-excpt-flags i))
             (implies (not (INST-fetch-error-detected-p i))
              (and (equal (DE-cntlv (DQ-DE1 (MA-DQ MA)))
                          (INST-cntlv i))
                   (equal (DE-rc (DQ-DE1 (MA-DQ MA)))
                          (rc (INST-word i)))
                   (equal (DE-ra (DQ-DE1 (MA-DQ MA)))
                          (ra (INST-word i)))
                   (equal (DE-rb (DQ-DE1 (MA-DQ MA)))
                          (rb (INST-word i)))
                   (equal (DE-im (DQ-DE1 (MA-DQ MA)))
                          (im (INST-word i)))
                   (equal (DE-br-target (DQ-DE1 (MA-DQ MA)))
                          (INST-br-target i))))
             (implies (INST-fetch-error-detected-p i)
                      (equal (DE-cntlv (DQ-DE1 (MA-DQ MA)))
                             (decode 0 (INST-br-predict? i))))))))

(defun DQ2-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))
       (implies (and (not (b1p (inst-specultv? i)))
                     (implies (b1p (INST-modified? i))
                              (b1p (INST-first-modified? i))))
                (equal (DE-pc (DQ-DE2 (MA-DQ MA))) (ISA-pc (INST-pre-ISA i))))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
        (and (equal (DE-excpt (DQ-DE2 (MA-DQ MA))) (INST-excpt-flags i))
             (implies (not (INST-fetch-error-detected-p i))
              (and (equal (DE-cntlv (DQ-DE2 (MA-DQ MA)))
                          (INST-cntlv i))
                   (equal (DE-rc (DQ-DE2 (MA-DQ MA)))
                          (rc (INST-word i)))
```

```
                  (equal (DE-ra (DQ-DE2 (MA-DQ MA)))
                         (ra (INST-word i)))
                  (equal (DE-rb (DQ-DE2 (MA-DQ MA)))
                         (rb (INST-word i)))
                  (equal (DE-im (DQ-DE2 (MA-DQ MA)))
                         (im (INST-word i)))
                  (equal (DE-br-target (DQ-DE2 (MA-DQ MA)))
                         (INST-br-target i))))
            (implies (INST-fetch-error-detected-p i)
                     (equal (DE-cntlv (DQ-DE2 (MA-DQ MA)))
                            (decode 0 (INST-br-predict? i))))))))

(defun DQ3-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (DE-valid? (DQ-DE3 (MA-DQ MA))))
       (implies (and (not (b1p (inst-specultv? i)))
                     (implies (b1p (INST-modified? i))
                              (b1p (INST-first-modified? i))))
                (equal (DE-pc (DQ-DE3 (MA-DQ MA))) (ISA-pc (INST-pre-ISA i))))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
        (and (equal (DE-excpt (DQ-DE3 (MA-DQ MA))) (INST-excpt-flags i))
             (implies (not (INST-fetch-error-detected-p i))
              (and (equal (DE-cntlv (DQ-DE3 (MA-DQ MA))) (INST-cntlv i))
                   (equal (DE-rc (DQ-DE3 (MA-DQ MA)))
                          (rc (INST-word i)))
                   (equal (DE-ra (DQ-DE3 (MA-DQ MA)))
                          (ra (INST-word i)))
                   (equal (DE-rb (DQ-DE3 (MA-DQ MA)))
                          (rb (INST-word i)))
                   (equal (DE-im (DQ-DE3 (MA-DQ MA)))
                          (im (INST-word i)))
                   (equal (DE-br-target (DQ-DE3 (MA-DQ MA)))
                          (INST-br-target i))))
             (implies (INST-fetch-error-detected-p i)
                      (equal (DE-cntlv (DQ-DE3 (MA-DQ MA)))
                             (decode 0 (INST-br-predict? i))))))))

(defun DQ-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (cond ((equal (INST-stg i) '(DQ 0))
         (DQ0-inst-inv i MA))
        ((equal (INST-stg i) '(DQ 1))
         (DQ1-inst-inv i MA))
        ((equal (INST-stg i) '(DQ 2))
         (DQ2-inst-inv i MA))
        ((equal (INST-stg i) '(DQ 3))
         (DQ3-inst-inv i MA))
        (t nil)))


(defun IU-RS0-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (RS-valid? (IU-RS0 (MA-IU MA))))
       ; This may not be provable.
       (equal (RS-tag (IU-RS0 (MA-IU MA))) (INST-tag i))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
         (and (b1p (INST-IU? i))
              (not (INST-excpt-detected-p i))
              (implies (b1p (logbit 3 (cntlv-operand (INST-cntlv i))))
                       (srname-p (INST-ra i)))
```

```
            (equal (RS-op (IU-RS0 (MA-IU MA))) (INST-IU-op? i))
            (implies (b1p (RS-ready1? (IU-RS0 (MA-IU MA))))
                     (equal (RS-val1 (IU-RS0 (MA-IU MA)))
                            (INST-src-val1 i)))
            (implies (and (b1p (RS-ready2? (IU-RS0 (MA-IU MA))))
                          (not (b1p (INST-IU-op? i))))
                     (equal (RS-val2 (IU-RS0 (MA-IU MA)))
                            (INST-src-val2 i)))))))

(defun IU-RS1-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (RS-valid? (IU-RS1 (MA-IU MA))))
       ; This tag field is related to the control of instructions.
       ; not the output of data.  Should be independent of speculative
       ; execution.
       (equal (RS-tag (IU-RS1 (MA-IU MA))) (INST-tag i))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
         (and (b1p (INST-IU? i))
              (not (INST-excpt-detected-p i))
              (implies (b1p (logbit 3 (cntlv-operand (INST-cntlv i))))
                       (srname-p (INST-ra i)))
              (equal (RS-op (IU-RS1 (MA-IU MA))) (INST-IU-op? i))
              (implies (b1p (RS-ready1? (IU-RS1 (MA-IU MA))))
                       (equal (RS-val1 (IU-RS1 (MA-IU MA)))
                              (INST-src-val1 i)))
              (implies (and (b1p (RS-ready2? (IU-RS1 (MA-IU MA))))
                            (not (b1p (INST-IU-op? i))))
                       (equal (RS-val2 (IU-RS1 (MA-IU MA)))
                              (INST-src-val2 i)))))))


(defun IU-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (cond ((equal (INST-stg i) '(IU RS0))
         (IU-RS0-inst-inv i MA))
        ((equal (INST-stg i) '(IU RS1))
         (IU-RS1-inst-inv i MA))
        (t nil)))

(defun BU-RS0-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (BU-RS-valid? (BU-RS0 (MA-BU MA))))
       (equal (BU-RS-tag (BU-RS0 (MA-BU MA))) (INST-tag i))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
         (and (b1p (INST-BU? i))
              (not (INST-excpt-detected-p i))
              (implies (b1p (BU-RS-ready? (BU-RS0 (MA-BU MA))))
                       (equal (BU-RS-val (BU-RS0 (MA-BU MA)))
                              (INST-src-val3 i)))))))

(defun BU-RS1-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (BU-RS-valid? (BU-RS1 (MA-BU MA))))
       (equal (BU-RS-tag (BU-RS1 (MA-BU MA))) (INST-tag i))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
         (and (b1p (INST-BU? i))
              (not (INST-excpt-detected-p i))
              (implies (b1p (BU-RS-ready? (BU-RS1 (MA-BU MA))))
                       (equal (BU-RS-val (BU-RS1 (MA-BU MA)))
```

```
                              (INST-src-val3 i)))))))

(defun BU-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (cond ((equal (INST-stg i) '(BU RS0))
         (BU-RS0-inst-inv i MA))
        ((equal (INST-stg i) '(BU RS1))
         (BU-RS1-inst-inv i MA))
        (t nil)))

(defun MU-RS0-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (RS-valid? (MU-RS0 (MA-MU MA))))
       (equal (RS-tag (MU-RS0 (MA-MU MA))) (INST-tag i))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
          (and (b1p (INST-MU? i))
               (not (INST-excpt-detected-p i))
               (equal (RS-op (MU-RS0 (MA-MU MA))) 0)
               (implies (b1p (RS-ready1? (MU-RS0 (MA-MU MA))))
                        (equal (RS-val1 (MU-RS0 (MA-MU MA)))
                               (INST-src-val1 i)))
               (implies (b1p (RS-ready2? (MU-RS0 (MA-MU MA))))
                        (equal (RS-val2 (MU-RS0 (MA-MU MA)))
                               (INST-src-val2 i)))))))

(defun MU-RS1-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (RS-valid? (MU-RS1 (MA-MU MA))))
       (equal (RS-tag (MU-RS1 (MA-MU MA))) (INST-tag i))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
          (and (b1p (INST-MU? i))
               (not (INST-excpt-detected-p i))
               (equal (RS-op (MU-RS1 (MA-MU MA))) 0)
               (implies (b1p (RS-ready1? (MU-RS1 (MA-MU MA))))
                        (equal (RS-val1 (MU-RS1 (MA-MU MA)))
                               (INST-src-val1 i)))
               (implies (b1p (RS-ready2? (MU-RS1 (MA-MU MA))))
                        (equal (RS-val2 (MU-RS1 (MA-MU MA)))
                               (INST-src-val2 i)))))))


(defun MU-lch1-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (MU-latch1-valid? (MU-lch1 (MA-MU MA))))
       (equal (MU-latch1-tag (MU-lch1 (MA-MU MA))) (INST-tag i))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
        (and (b1p (INST-MU? i))
             (not (INST-excpt-detected-p i))
             (equal (MU-latch1-data (MU-lch1 (MA-MU MA)))
                    (ML1-output (INST-src-val1 i) (INST-src-val2 i)))))))

(defun MU-lch2-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (MU-latch2-valid? (MU-lch2 (MA-MU MA))))
       (equal (MU-latch2-tag (MU-lch2 (MA-MU MA))) (INST-tag i))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
        (and (b1p (INST-MU? i))
             (not (INST-excpt-detected-p i))
```

```
            (equal (MU-latch2-data (MU-lch2 (MA-MU MA)))
                   (ML2-output
                    (ML1-output (INST-src-val1 i) (INST-src-val2 i))))))))

(defun MU-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (cond ((equal (INST-stg i) '(MU RS0))
         (MU-RS0-inst-inv i MA))
        ((equal (INST-stg i) '(MU RS1))
         (MU-RS1-inst-inv i MA))
        ((equal (INST-stg i) '(MU lch1))
         (MU-lch1-inst-inv i MA))
        ((equal (INST-stg i) '(MU lch2))
         (MU-lch2-inst-inv i MA))
        (t nil)))

(defun LSU-RS0-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (LSU-RS-valid? (LSU-RS0 (MA-LSU MA))))
       (equal (LSU-RS-tag (LSU-RS0 (MA-LSU MA))) (INST-tag i))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
         (and (b1p (INST-LSU? i))
              (not (INST-excpt-detected-p i))
              (equal (LSU-RS-op (LSU-RS0 (MA-LSU MA)))
                     (INST-LSU-op? i))
              (equal (LSU-RS-ld-st? (LSU-RS0 (MA-LSU MA)))
                     (INST-ld-st? i))
              (implies (b1p (INST-LSU-op? i))
                       (b1p (LSU-RS-rdy1? (LSU-RS0 (MA-LSU MA)))))
              (implies (b1p (LSU-RS-rdy1? (LSU-RS0 (MA-LSU MA))))
                       (equal (LSU-RS-val1 (LSU-RS0 (MA-LSU MA)))
                              (INST-src-val1 i)))
              (implies (b1p (LSU-RS-rdy2? (LSU-RS0 (MA-LSU MA))))
                       (equal (LSU-RS-val2 (LSU-RS0 (MA-LSU MA)))
                              (INST-src-val2 i)))
              (implies (b1p (LSU-RS-rdy3? (LSU-RS0 (MA-LSU MA))))
                       (equal (LSU-RS-val3 (LSU-RS0 (MA-LSU MA)))
                              (INST-src-val3 i)))))))

(defun LSU-RS1-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (LSU-RS-valid? (LSU-RS1 (MA-LSU MA))))
       (equal (LSU-RS-tag (LSU-RS1 (MA-LSU MA))) (INST-tag i))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
         (and (b1p (INST-LSU? i))
              (not (INST-excpt-detected-p i))
              (equal (LSU-RS-op (LSU-RS1 (MA-LSU MA)))
                     (INST-LSU-op? i))
              (equal (LSU-RS-ld-st? (LSU-RS1 (MA-LSU MA)))
                     (INST-ld-st? i))
              (implies (b1p (INST-LSU-op? i))
                       (b1p (LSU-RS-rdy1? (LSU-RS1 (MA-LSU MA)))))
              (implies (b1p (LSU-RS-rdy1? (LSU-RS1 (MA-LSU MA))))
                       (equal (LSU-RS-val1 (LSU-RS1 (MA-LSU MA)))
                              (INST-src-val1 i)))
              (implies (b1p (LSU-RS-rdy2? (LSU-RS1 (MA-LSU MA))))
                       (equal (LSU-RS-val2 (LSU-RS1 (MA-LSU MA)))
                              (INST-src-val2 i)))
              (implies (b1p (LSU-RS-rdy3? (LSU-RS1 (MA-LSU MA))))
                       (equal (LSU-RS-val3 (LSU-RS1 (MA-LSU MA)))
```

```
                              (INST-src-val3 i)))))))

(defun LSU-rbuf-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (rbuf-valid? (LSU-rbuf (MA-LSU MA))))
       (equal (rbuf-tag (LSU-rbuf (MA-LSU MA))) (INST-tag i))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
        (and (b1p (INST-load? i))
             (not (INST-excpt-detected-p i))
             (equal (rbuf-addr (LSU-rbuf (MA-LSU MA)))
                    (INST-load-addr i))))))

(defun LSU-lch-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (LSU-latch-valid? (LSU-lch (MA-LSU MA))))
       (equal (LSU-latch-tag (LSU-lch (MA-LSU MA))) (INST-tag i))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
        (and (b1p (INST-load? i))
             (not (INST-fetch-error-detected-p i))
             (not (INST-decode-error-detected-p i))
             (equal (LSU-latch-excpt (LSU-lch (MA-LSU MA)))
                    (INST-excpt-flags i))
             (implies (not (INST-load-accs-error-detected-p i))
                      (equal (LSU-latch-val (LSU-lch (MA-LSU MA)))
                             (INST-dest-val i)))))))

(defun LSU-wbuf0-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
       (not (b1p (wbuf-complete? (LSU-wbuf0 (MA-LSU MA)))))
       (not (b1p (wbuf-commit? (LSU-wbuf0 (MA-LSU MA)))))
       (equal (wbuf-tag (LSU-wbuf0 (MA-LSU MA))) (INST-tag i))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
        (and (b1p (INST-store? i))
             (not (INST-excpt-detected-p i))
             (equal (wbuf-addr (LSU-wbuf0 (MA-LSU MA))) (INST-store-addr i))
             (equal (wbuf-val (LSU-wbuf0 (MA-LSU MA))) (INST-src-val3 i))))))

(defun LSU-wbuf1-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
       (not (b1p (wbuf-complete? (LSU-wbuf1 (MA-LSU MA)))))
       (not (b1p (wbuf-commit? (LSU-wbuf1 (MA-LSU MA)))))
       (equal (wbuf-tag (LSU-wbuf1 (MA-LSU MA))) (INST-tag i))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
        (and (b1p (INST-store? i))
             (not (INST-excpt-detected-p i))
             (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA))) (INST-store-addr i))
             (equal (wbuf-val (LSU-wbuf1 (MA-LSU MA))) (INST-src-val3 i))))))

(defun LSU-wbuf0-lch-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
       (b1p (wbuf-complete? (LSU-wbuf0 (MA-LSU MA))))
       (not (b1p (wbuf-commit? (LSU-wbuf0 (MA-LSU MA)))))
       (b1p (LSU-latch-valid? (LSU-lch (MA-LSU MA))))
       (equal (wbuf-tag (LSU-wbuf0 (MA-LSU MA))) (INST-tag i))
       (equal (LSU-latch-tag (LSU-lch (MA-LSU MA))) (INST-tag i))
```

```
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
        (and (b1p (INST-store? i))
             (not (INST-fetch-error-detected-p i))
             (not (INST-decode-error-detected-p i))
             (equal (wbuf-addr (LSU-wbuf0 (MA-LSU MA))) (INST-store-addr i))
             (equal (wbuf-val (LSU-wbuf0 (MA-LSU MA))) (INST-src-val3 i))
             (equal (LSU-latch-excpt (LSU-lch (MA-LSU MA)))
                    (INST-excpt-flags i))))))

(defun LSU-wbuf1-lch-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
       (b1p (wbuf-complete? (LSU-wbuf1 (MA-LSU MA))))
       (not (b1p (wbuf-commit? (LSU-wbuf1 (MA-LSU MA)))))
       (b1p (LSU-latch-valid? (LSU-lch (MA-LSU MA))))
       (equal (wbuf-tag (LSU-wbuf1 (MA-LSU MA))) (INST-tag i))
       (equal (LSU-latch-tag (LSU-lch (MA-LSU MA))) (INST-tag i))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
        (and (b1p (INST-store? i))
             (not (INST-fetch-error-detected-p i))
             (not (INST-decode-error-detected-p i))
             (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA))) (INST-store-addr i))
             (equal (wbuf-val (LSU-wbuf1 (MA-LSU MA))) (INST-src-val3 i))
             (equal (LSU-latch-excpt (LSU-lch (MA-LSU MA)))
                    (INST-excpt-flags i))))))

(defun LSU-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (cond ((equal (INST-stg i) '(LSU RS0))
         (LSU-RS0-inst-inv i MA))
        ((equal (INST-stg i) '(LSU RS1))
         (LSU-RS1-inst-inv i MA))
        ((equal (INST-stg i) '(LSU rbuf))
         (LSU-rbuf-inst-inv i MA))
        ((equal (INST-stg i) '(LSU lch))
         (LSU-lch-inst-inv i MA))
        ((equal (INST-stg i) '(LSU wbuf0))
         (LSU-wbuf0-inst-inv i MA))
        ((equal (INST-stg i) '(LSU wbuf1))
         (LSU-wbuf1-inst-inv i MA))
        ((equal (INST-stg i) '(LSU wbuf0 lch))
         (LSU-wbuf0-lch-inst-inv i MA))
        ((equal (INST-stg i) '(LSU wbuf1 lch))
         (LSU-wbuf1-lch-inst-inv i MA))
        (t nil)))

(defun execute-inst-robe-inv (i robe)
  (declare (xargs :guard (and (INST-p i) (rob-entry-p robe))))
  (and (b1p (robe-valid? robe))
       (not (b1p (robe-complete? robe)))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
             (equal (robe-pc robe) (ISA-pc (INST-pre-ISA i))))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i)))
                     (or (b1p (INST-fetch-error? i))
                         (b1p (INST-decode-error? i))))
                (equal (robe-excpt robe) (INST-excpt-flags i)))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i)))
```

```
                    (INST-fetch-error-detected-p i))
               (and (equal (robe-branch? robe) 0)
                    (equal (robe-sync? robe) 0)
                    (equal (robe-rfeh? robe) 0)))
      (implies (and (not (b1p (inst-specultv? i)))
                    (not (b1p (INST-modified? i)))
                    (not (INST-fetch-error-detected-p i)))
       (and (equal (robe-wb? robe) (INST-wb? i))
            (equal (robe-wb-sreg? robe) (INST-wb-sreg? i))
            (equal (robe-sync? robe) (INST-sync? i))
            (equal (robe-branch? robe) (INST-BU? i))
            (equal (robe-rfeh? robe) (INST-rfeh? i))
            (equal (robe-br-predict? robe) (INST-br-predict? i))
            (implies (INST-writeback-p i)
                     (equal (robe-dest robe) (INST-dest-reg i)))
            (implies (b1p (INST-bu? i))
                     (equal (robe-val robe) (INST-br-target i)))))))

(in-theory (disable IU-inst-inv MU-inst-inv BU-inst-inv LSU-inst-inv
                    execute-inst-robe-inv))

(defun execute-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (cond ((IU-stg-p (INST-stg i))
              (IU-inst-inv i MA))
             ((MU-stg-p (INST-stg i))
              (MU-inst-inv i MA))
             ((BU-stg-p (INST-stg i))
              (BU-inst-inv i MA))
             ((LSU-stg-p (INST-stg i))
              (LSU-inst-inv i MA))
             (t nil))
       (execute-inst-robe-inv i (nth-robe (INST-tag i) (MA-rob MA)))))

(defun complete-normal-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (implies (and (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (INST-fetch-error-detected-p i))
                (not (INST-decode-error-detected-p i)))
           (not (b1p (INST-store? i)))))

(defun complete-wbuf0-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
       (b1p (wbuf-complete? (LSU-wbuf0 (MA-LSU MA))))
       (not (b1p (wbuf-commit? (LSU-wbuf0 (MA-LSU MA)))))
       (equal (wbuf-tag (LSU-wbuf0 (MA-LSU MA))) (INST-tag i))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
        (and (b1p (INST-store? i))
             (not (INST-fetch-error-detected-p i))
             (not (INST-decode-error-detected-p i))
             (equal (wbuf-addr (LSU-wbuf0 (MA-LSU MA))) (INST-store-addr i))
             (equal (wbuf-val (LSU-wbuf0 (MA-LSU MA))) (INST-src-val3 i))))))

(defun complete-wbuf1-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
       (b1p (wbuf-complete? (LSU-wbuf1 (MA-LSU MA))))
       (not (b1p (wbuf-commit? (LSU-wbuf1 (MA-LSU MA)))))
       (equal (wbuf-tag (LSU-wbuf1 (MA-LSU MA))) (INST-tag i))
```

```
      (implies (and (not (b1p (inst-specultv? i)))
                    (not (b1p (INST-modified? i))))
       (and (b1p (INST-store? i))
            (not (INST-fetch-error-detected-p i))
            (not (INST-decode-error-detected-p i))
            (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA))) (INST-store-addr i))
            (equal (wbuf-val (LSU-wbuf1 (MA-LSU MA))) (INST-src-val3 i))))))

(defun complete-inst-robe-inv (i robe)
  (declare (xargs :guard (and (INST-p i) (rob-entry-p robe))))
  (and (b1p (robe-valid? robe))
       (b1p (robe-complete? robe))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
                (and (equal (robe-excpt robe) (INST-excpt-flags i))
                     (equal (robe-pc robe) (ISA-pc (INST-pre-ISA i)))))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i)))
                     (INST-fetch-error-detected-p i))
                (and (equal (robe-branch? robe) 0)
                     (equal (robe-sync? robe) 0)
                     (equal (robe-rfeh? robe) 0)))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i)))
                     (not (INST-fetch-error-detected-p i)))
        (and (equal (robe-wb? robe) (INST-wb? i))
             (equal (robe-wb-sreg? robe) (INST-wb-sreg? i))
             (equal (robe-sync? robe) (INST-sync? i))
             (equal (robe-branch? robe) (INST-BU? i))
             (equal (robe-rfeh? robe) (INST-rfeh? i))
             (equal (robe-br-predict? robe) (INST-br-predict? i))
             (implies (b1p (INST-BU? i))
                      (equal (robe-br-actual? robe) (INST-branch-taken? i)))
             (implies (INST-writeback-p i)
                      (equal (robe-dest robe) (INST-dest-reg i)))
             (implies (and (INST-writeback-p i)
                           (not (INST-decode-error-detected-p i))
                           (not (INST-data-accs-error-detected-p i)))
                      (equal (robe-val robe) (INST-dest-val i)))
             (implies (b1p (INST-bu? i))
                      (equal (robe-val robe) (INST-br-target i)))))))

(defun complete-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (cond ((equal (INST-stg i) '(complete))
              (complete-normal-inst-inv i MA))
             ((equal (INST-stg i) '(complete wbuf0))
              (complete-wbuf0-inst-inv i MA))
             ((equal (INST-stg i) '(complete wbuf1))
              (complete-wbuf1-inst-inv i MA))
             (t nil))
       (complete-inst-robe-inv i (nth-robe (INST-tag i) (MA-rob MA)))))

(defun commit-wbuf0-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
       (not (b1p (INST-excpt? i)))
       (b1p (wbuf-complete? (LSU-wbuf0 (MA-LSU MA))))
       (b1p (wbuf-commit? (LSU-wbuf0 (MA-LSU MA))))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
        (and (b1p (INST-store? i))
```

```
            (equal (wbuf-addr (LSU-wbuf0 (MA-LSU MA))) (INST-store-addr i))
            (equal (wbuf-val (LSU-wbuf0 (MA-LSU MA))) (INST-src-val3 i))))))

(defun commit-wbuf1-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (and (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
       (not (b1p (INST-excpt? i)))
       (b1p (wbuf-complete? (LSU-wbuf1 (MA-LSU MA))))
       (b1p (wbuf-commit? (LSU-wbuf1 (MA-LSU MA))))
       (implies (and (not (b1p (inst-specultv? i)))
                     (not (b1p (INST-modified? i))))
        (and (b1p (INST-store? i))
             (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA))) (INST-store-addr i))
             (equal (wbuf-val (LSU-wbuf1 (MA-LSU MA))) (INST-src-val3 i))))))

(defun commit-inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (cond ((equal (INST-stg i) '(commit wbuf0))
         (commit-wbuf0-inst-inv i MA))
        ((equal (INST-stg i) '(commit wbuf1))
         (commit-wbuf1-inst-inv i MA))
        (t nil)))

; The predicates above are the stage-dependent condition that
; individual instructions should satisfy at the stage.  The following
; predicate inst-inv is true for any instruction i regardless of its
; stage. Basically, it is the conjunction of all the stage-dependent
; conditions.
(defun inst-inv (i MA)
  (declare (xargs :guard (and (INST-p i) (MA-state-p MA))))
  (cond ((IFU-stg-p (INST-stg i))
         (IFU-inst-inv i MA))
        ((DQ-stg-p (INST-stg i))
         (DQ-inst-inv i MA))
        ((execute-stg-p (INST-stg i))
         (execute-inst-inv i MA))
        ((complete-stg-p (INST-stg i))
         (complete-inst-inv i MA))
        ((commit-stg-p (INST-stg i))
         (commit-inst-inv i MA))
        (t ; retire
         T)))
(deflabel end-inst-inv-def)

(deftheory inst-inv-def
    (set-difference-theories (universal-theory 'end-inst-inv-def)
                             (universal-theory 'begin-inst-inv-def)))

(defun trace-INST-inv (trace MA)
  (declare (xargs :guard (and (INST-listp trace) (MA-state-p MA))))
  (if (endp trace)
      T
      (and (inst-inv (car trace) MA)
           (trace-INST-inv (cdr trace) MA))))

; The property that intermediate results in the pipeline are correct.
(defun MT-INST-inv (MT MA)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA))))
  (trace-INST-inv (MT-trace MT) MA))

; ROB-flg, ROB-head and ROB-tail determines which entries in ROB are
; occupied.  If ROB-flg is on, ROB-tail is less than or equal to ROB-head,
```

```
; and a ROB entry is occupied if its index is less than ROB-tail, or it is
; larger than or equal to ROB-head.  If ROB-flg is off, ROB-tail is
; larger than or equal to ROB-head.  An entry is occupied if
; its index is less than ROB-tail and larger or equal to ROB-head.
; Note:
; When an branch instruction is in a ROB entry, it should not have
; caused an exception.  However, this turned out to be difficult to
; verify, so we commented it out and postponed the verification.  Since
; we have completed the verification it was not a necessary condition
; for the verification of our correctness criterion.
(defun consistent-robe-p (robe idx ROB)
  (declare (xargs :guard (and (rob-entry-p robe)
                              (rob-index-p idx) (rob-p ROB))))
  (and (equal (robe-valid? robe)
              (if (b1p (ROB-flg ROB))
                  (if (or (<= (ROB-head ROB) idx) (< idx (ROB-tail ROB)))
                      1 0)
                  (if (and (<= (ROB-head ROB) idx) (< idx (ROB-tail ROB)))
                      1 0)))
       (implies (b1p (robe-valid? robe))
;                     (not (b1p (b-and (robe-branch? robe)
;                                      (excpt-raised? (robe-excpt robe)))))
                (and (not (b1p (b-and (robe-branch? robe) (robe-sync? robe))))
                     (not (b1p (b-and (robe-wb? robe) (robe-sync? robe))))))))

(defun consistent-rob-entries-p (entries idx ROB)
  (declare (xargs :guard (and (rob-p rob) (rob-index-p idx)
                              (ROBE-listp entries))))
  (if (endp entries)
      T
    (and (consistent-robe-p (car entries) idx ROB)
         (consistent-rob-entries-p (cdr entries) (rob-index (1+ idx)) ROB))))

(defun consistent-rob-flg-p (ROB)
  (declare (xargs :guard (rob-p rob)))
  (if (b1p (ROB-flg ROB))
      (<= (ROB-tail ROB) (ROB-head ROB))
      (<= (ROB-head ROB) (ROB-tail ROB))))

(defun consistent-rob-p (ROB)
  (declare (xargs :guard (rob-p rob)))
  (and (consistent-rob-entries-p (ROB-entries ROB) 0 ROB)
       (consistent-rob-flg-p ROB)))

(defun consistent-cntlv-p (cntlv)
  (declare (xargs :guard (cntlv-p cntlv)))
  (not (b1p (bs-ior (b-and (logbit 0 (cntlv-exunit cntlv))
                           (logbit 1 (cntlv-exunit cntlv)))
                    (b-and (logbit 0 (cntlv-exunit cntlv))
                           (logbit 2 (cntlv-exunit cntlv)))
                    (b-and (logbit 0 (cntlv-exunit cntlv))
                           (logbit 3 (cntlv-exunit cntlv)))
                    (b-and (logbit 0 (cntlv-exunit cntlv))
                           (logbit 4 (cntlv-exunit cntlv)))
                    (b-and (logbit 1 (cntlv-exunit cntlv))
                           (logbit 2 (cntlv-exunit cntlv)))
                    (b-and (logbit 1 (cntlv-exunit cntlv))
                           (logbit 3 (cntlv-exunit cntlv)))
                    (b-and (logbit 1 (cntlv-exunit cntlv))
                           (logbit 4 (cntlv-exunit cntlv)))
                    (b-and (logbit 2 (cntlv-exunit cntlv))
                           (logbit 3 (cntlv-exunit cntlv)))
```

```
                    (b-and (logbit 2 (cntlv-exunit cntlv))
                           (logbit 4 (cntlv-exunit cntlv)))
                    (b-and (logbit 3 (cntlv-exunit cntlv))
                           (logbit 4 (cntlv-exunit cntlv)))
                    (b-and (logbit 3 (cntlv-exunit cntlv))
                           (cntlv-sync? cntlv))
                    (b-and (cntlv-wb? cntlv)
                           (cntlv-sync? cntlv))))))

(defun consistent-DQ-cntlv-p (DQ)
  (declare (xargs :guard (DQ-p DQ)))
  (and (implies (b1p (DE-valid? (DQ-DE0 DQ)))
                (consistent-cntlv-p (DE-cntlv (DQ-DE0 DQ))))
       (implies (b1p (DE-valid? (DQ-DE1 DQ)))
                (consistent-cntlv-p (DE-cntlv (DQ-DE1 DQ))))
       (implies (b1p (DE-valid? (DQ-DE2 DQ)))
                (consistent-cntlv-p (DE-cntlv (DQ-DE2 DQ))))
       (implies (b1p (DE-valid? (DQ-DE3 DQ)))
                (consistent-cntlv-p (DE-cntlv (DQ-DE3 DQ))))))

(defun consistent-LSU-p (LSU)
  (declare (xargs :guard (load-store-unit-p LSU)))
  (and (implies (and (b1p (rbuf-valid? (LSU-rbuf LSU)))
                     (b1p (rbuf-wbuf0? (LSU-rbuf LSU))))
                (b1p (wbuf-valid? (LSU-wbuf0 LSU))))
       (implies (and (b1p (rbuf-valid? (LSU-rbuf LSU)))
                     (b1p (rbuf-wbuf1? (LSU-rbuf LSU))))
                (b1p (wbuf-valid? (LSU-wbuf1 LSU))))
       (implies (b1p (wbuf-valid? (LSU-wbuf1 LSU)))
                (b1p (wbuf-valid? (LSU-wbuf0 LSU))))))


; A miscellaneous conditions.
(defun consistent-MA-p (MA)
  (declare (xargs :guard (MA-state-p MA)))
  (and (consistent-DQ-cntlv-p (MA-DQ MA))
       (consistent-rob-p (MA-ROB MA))
       (consistent-LSU-p (MA-LSU MA))))

(defun no-dispatched-inst-p (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace)
      T
      (and (not (dispatched-p (car trace)))
           (no-dispatched-inst-p (cdr trace)))))

(defun no-commit-inst-p (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace)
      T
      (and (not (committed-p (car trace)))
           (no-commit-inst-p (cdr trace)))))

(defun in-order-trace-p (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace)
      T
      (and (cond ((IFU-stg-p (INST-stg (car trace))) (endp (cdr trace)))
                 ((not (dispatched-p (car trace)))
                  (no-dispatched-inst-p (cdr trace)))
                 ((not (committed-p (car trace)))
                  (no-commit-inst-p (cdr trace)))
```

```
                (t
                 t))
           (in-order-trace-p (cdr trace)))))

; Instructions are dispatched and committed in order.
(defun in-order-dispatch-commit-p (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (in-order-trace-p (MT-trace MT)))

(defun in-order-DQ-trace-p (trace idx)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace)
      t
      (if (IFU-stg-p (INST-stg (car trace)))
          t
          (if (DQ-stg-p (INST-stg (car trace)))
              (and (equal (DQ-stg-idx (INST-stg (car trace))) idx)
                   (in-order-DQ-trace-p (cdr trace) (+ 1 idx)))
              (in-order-DQ-trace-p (cdr trace) idx)))))

; The dispatch queue is a FIFO queue.
(defun in-order-DQ-p (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (in-order-DQ-trace-p (MT-trace MT) 0))

(defun no-INST-at-wbuf0-p (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace) T
      (and (not (wbuf0-stg-p (INST-stg (car trace))))
           (no-INST-at-wbuf0-p (cdr trace)))))

(defun no-INST-at-wbuf1-p (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace) T
      (and (not (wbuf1-stg-p (INST-stg (car trace))))
           (no-INST-at-wbuf1-p (cdr trace)))))

(defun no-INST-at-wbuf-p (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace) T
      (and (not (wbuf-stg-p (INST-stg (car trace))))
           (no-INST-at-wbuf-p (cdr trace)))))

; The instructions in the write-buffer should be in the correct order;
; the instruction in wbuf0 should precede the instruction in wbuf1.
(defun in-order-WB-trace-p (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace) T
      (and (implies (wbuf1-stg-p (INST-stg (car trace)))
                    (no-INST-at-wbuf0-p (cdr trace)))
           (in-order-WB-trace-p (cdr trace)))))

(defun no-retired-store-p (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace) T
      (and (not (and (retire-stg-p (INST-stg (car trace)))
                     (b1p (INST-store? (car trace)))))
           (no-retired-store-p (cdr trace)))))

; Store instructions should retire in program order.
(defun in-order-WB-retire-p (trace)
  (declare (xargs :guard (INST-listp trace)))
```

```
  (if (endp trace) T
      (and (implies (wbuf0-stg-p (INST-stg (car trace)))
                    (no-retired-store-p (cdr trace)))
           (in-order-WB-retire-p (cdr trace)))))


(defun LSU-issued-stg-p (stg)
  (declare (xargs :guard (stage-p stg)))
  (or (equal stg '(LSU wbuf0))
      (equal stg '(LSU wbuf1))
      (equal stg '(LSU rbuf))
      (equal stg '(LSU wbuf0 lch))
      (equal stg '(LSU wbuf1 lch))
      (equal stg '(LSU lch))
      (equal stg '(complete wbuf0))
      (equal stg '(complete wbuf1))
      (equal stg '(commit wbuf0))
      (equal stg '(commit wbuf1))))

(defun no-issued-LSU-inst-p (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace) T
      (and (not (LSU-issued-stg-p (INST-stg (car trace))))
           (no-issued-LSU-inst-p (cdr trace)))))

; Instructions in LSU should be issued in the program order.
(defun in-order-LSU-issue-p (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace) T
      (and (implies (or (equal (INST-stg (car trace)) '(LSU RS0))
                        (equal (INST-stg (car trace)) '(LSU RS1)))
                    (no-issued-LSU-inst-p (cdr trace)))
           (in-order-LSU-issue-p (cdr trace)))))

; Instructions in the reservations stations attached to LSU should be in the
; correct order, depending on the flag LSU-rs1-head.
(defun in-order-LSU-RS-p (trace MA)
  (declare (xargs :guard (and (INST-listp trace) (MA-state-p MA))))
  (if (endp trace) T
      (and (cond ((and (b1p (LSU-rs1-head? (MA-LSU MA)))
                       (equal (INST-stg (car trace)) '(LSU RS0)))
                  (no-INST-at-stg-in-trace '(LSU RS1) (cdr trace)))
                 ((and (not (b1p (LSU-rs1-head? (MA-LSU MA))))
                       (equal (INST-stg (car trace)) '(LSU RS1)))
                  (no-INST-at-stg-in-trace '(LSU RS0) (cdr trace)))
                 (t t))
           (in-order-LSU-RS-p (cdr trace) MA))))

; The following predicate is true, if the instruction in wbuf0 appears
; earlier in trace than the instruction at rbuf.  Similarly, following
; predicates define the order of instructions in the read and write
; buffers.
(defun in-order-wbuf0-rbuf-p (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace) t
      (and (implies  (equal (INST-stg (car trace)) '(LSU rbuf))
                     (no-INST-at-wbuf0-p (cdr trace)))
           (in-order-wbuf0-rbuf-p (cdr trace)))))

(defun in-order-rbuf-wbuf0-p (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace) t
```

```
      (and (implies  (wbuf0-stg-p (INST-stg (car trace)))
                     (no-INST-at-stg-in-trace '(LSU rbuf) (cdr trace)))
           (in-order-rbuf-wbuf0-p (cdr trace)))))

(defun in-order-wbuf1-rbuf-p (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace) t
      (and (implies  (equal (INST-stg (car trace)) '(LSU rbuf))
                     (no-INST-at-wbuf1-p (cdr trace)))
           (in-order-wbuf1-rbuf-p (cdr trace)))))

(defun in-order-rbuf-wbuf1-p (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace) t
      (and (implies  (wbuf1-stg-p (INST-stg (car trace)))
                     (no-INST-at-stg-in-trace '(LSU rbuf) (cdr trace)))
           (in-order-rbuf-wbuf1-p (cdr trace)))))


; Rbuf-wbuf0? records whether the instruction stored in wbuf0 precedes the
; instruction stored in rbuf.  According to the flag, the instructions should
; be recorded in MT in the correct order.  Similarly with rbuf-wbuf1?.
(defun in-order-load-store-p (MT MA)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA))))
  (and (implies (and (b1p (rbuf-valid? (LSU-rbuf (MA-LSU MA))))
                     (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
                (if (b1p (rbuf-wbuf0? (LSU-rbuf (MA-LSU MA))))
                    (in-order-wbuf0-rbuf-p (MT-trace MT))
                    (in-order-rbuf-wbuf0-p (MT-trace MT))))
       (implies (and (b1p (rbuf-valid? (LSU-rbuf (MA-LSU MA))))
                     (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                (if (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA))))
                    (in-order-wbuf1-rbuf-p (MT-trace MT))
                    (in-order-rbuf-wbuf1-p (MT-trace MT))))))


; This predicate checks instructions are stored in LSU in a certain order.
; The violation of the constraints causes incorrect memory operations.
(defun in-order-LSU-inst-p (MT MA)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA))))
  (and (in-order-WB-trace-p (MT-trace MT))
       (in-order-LSU-issue-p (MT-trace MT))
       (in-order-LSU-RS-p (MT-trace MT) MA)
       (in-order-WB-retire-p (MT-trace MT))
       (in-order-load-store-p MT MA)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Various functions for register modifiers
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Definition of Last Register Modifiers in the ROB.  The ROB records
; all instructions that are dispatched but not yet committed.  When an
; instruction is dispatched, its destination register does not contain
; the correct register value as a source value of following
; instructions.  Each register records which instruction should
; eventually writes the correct value of register in the future.
; (This information is stored in the register reference table.)  We
; call this instruction as the last register modifier in the ROB.  The
; last register modifier I of register idx in the ROB can be
; characterized by three conditions.
; 1. I is dispatched but not yet committed.
; 2. I writes back its result to the register idx.
; 3. I is the last instruction in program order of all such
```

```
; instructions.
; We define a function that tells whether there is any modifier of
; register idx in the ROB, and a function that returns the last
; register modifier.
;
; Instruction i modifies register idx.
(defun reg-modifier-p (idx i)
  (declare (xargs :guard (and (rname-p idx) (INST-p i))))
  (and (b1p (INST-wb? i)) (not (b1p (INST-wb-sreg? i)))
       (equal (INST-dest-reg i) idx)))

; Instruction i modifies special register idx.
(defun sreg-modifier-p (idx i)
  (declare (xargs :guard (and (rname-p idx) (INST-p i))))
  (and (b1p (INST-wb? i))  (b1p (INST-wb-sreg? i))
       (equal (INST-dest-reg i) idx)))


; trace-exist-LRM-in-ROB-p is used to define exist-LRM-in-ROB-p.
(defun trace-exist-LRM-in-ROB-p (idx trace)
  (declare (xargs :guard (and (rname-p idx) (INST-listp trace))))
  (if (endp trace)
      nil
      (if (and (dispatched-p (car trace))
               (not (committed-p (car trace)))
               (reg-modifier-p idx (car trace)))
          t
          (trace-exist-LRM-in-ROB-p idx (cdr trace)))))

; (exist-LRM-in-ROB-p idx MT) is true if there exists a register
; modifier of register idx in the ROB in the state represented by
; MT. (If an instruction is dispatched and not committed, it is
; assigned to an entry in the ROB.)
(defun exist-LRM-in-ROB-p (idx MT)
  (declare (xargs :guard (and (rname-p idx) (MAETT-p MT))))
  (trace-exist-LRM-in-ROB-p idx (MT-trace MT)))

(defun trace-exist-LSRM-in-ROB-p (idx trace)
  (declare (xargs :guard (and (rname-p idx) (INST-listp trace))))
  (if (endp trace)
      nil
      (if (and (dispatched-p (car trace))
               (not (committed-p (car trace)))
               (sreg-modifier-p idx (car trace)))
          t
          (trace-exist-LSRM-in-ROB-p idx (cdr trace)))))

; (exist-LSRM-in-ROB-p idx MT) is t when there exists an register
; modifier of register idx in ROB in the state represented by MT.
(defun exist-LSRM-in-ROB-p (idx MT)
  (declare (xargs :guard (and (rname-p idx) (MAETT-p MT))))
  (trace-exist-LSRM-in-ROB-p idx (MT-trace MT)))

(defun trace-LRM-in-ROB (idx trace )
  (declare (xargs :guard (and (rname-p idx) (INST-listp trace))))
  (if (endp trace)
      nil
      (if (and (dispatched-p (car trace))
               (not (committed-p (car trace)))
               (reg-modifier-p idx (car trace))
               (not (trace-exist-LRM-in-ROB-p idx (cdr trace))))
          (car trace)
```

```
          (trace-LRM-in-ROB idx (cdr trace)))))

; The last register modifier of register idx in ROB.
(defun LRM-in-ROB (idx MT)
  (declare (xargs :guard (and (rname-p idx) (MAETT-p MT))))
  (trace-LRM-in-ROB idx (MT-trace MT)))

(in-theory (disable LRM-in-ROB))

(defun trace-LSRM-in-ROB (idx trace )
  (declare (xargs :guard (and (rname-p idx) (INST-listp trace))))
  (if (endp trace)
      nil
      (if (and (dispatched-p (car trace))
               (not (committed-p (car trace)))
               (sreg-modifier-p idx (car trace))
               (not (trace-exist-LSRM-in-ROB-p idx (cdr trace))))
          (car trace)
          (trace-LSRM-in-ROB idx (cdr trace)))))

; The last register modifier of special register idx in the ROB in
; the state represented by MT.
(defun LSRM-in-ROB (idx MT)
  (declare (xargs :guard (and (rname-p idx) (MAETT-p MT))))
  (trace-LSRM-in-ROB idx (MT-trace MT)))

(in-theory (disable LSRM-in-ROB))

(defun trace-exist-LRM-before-p (i idx trace)
  (declare (xargs :guard (and (INST-p i) (rname-p idx)
                              (INST-listp trace))))
  (if (endp trace)
      nil
      (if (equal (car trace) i)
          nil
          (if (reg-modifier-p idx (car trace))
              t
              (trace-exist-LRM-before-p i idx (cdr trace))))))


(defun trace-exist-LSRM-before-p (i idx trace)
  (declare (xargs :guard (and (INST-p i) (rname-p idx)
                              (INST-listp trace))))
  (if (endp trace)
      nil
      (if (equal (car trace) i)
          nil
          (if (sreg-modifier-p idx (car trace))
              t
              (trace-exist-LSRM-before-p i idx (cdr trace))))))

; If there is an uncommitted modifier of register idx before instruction i,
; exist-LRM-before-p returns t.
(defun exist-LRM-before-p (i idx MT)
  (declare (xargs :guard (and (INST-p i) (rname-p idx) (MAETT-p MT))))
  (trace-exist-LRM-before-p i idx (MT-trace MT)))

; If there is an uncommitted modifier of special register idx before
; instruction i, exist-LRM-before-p returns t.
(defun exist-LSRM-before-p (i idx MT)
  (declare (xargs :guard (and (INST-p i) (rname-p idx) (MAETT-p MT))))
  (trace-exist-LSRM-before-p i idx (MT-trace MT)))
```

```
(in-theory (disable exist-LRM-before-p
                    exist-LSRM-before-p))

(defun trace-exist-uncommitted-LRM-before-p (i idx trace)
  (declare (xargs :guard (and (INST-p i) (rname-p idx)
                              (INST-listp trace))))
  (if (endp trace)
      nil
      (if (equal (car trace) i)
          nil
          (if (and (reg-modifier-p idx (car trace))
                   (not (committed-p (car trace))))
              t
              (trace-exist-uncommitted-LRM-before-p i idx (cdr trace))))))

(defun trace-exist-uncommitted-LSRM-before-p (i idx trace)
  (declare (xargs :guard (and (INST-p i) (rname-p idx)
                              (INST-listp trace))))
  (if (endp trace)
      nil
      (if (equal (car trace) i)
          nil
          (if (and (sreg-modifier-p idx (car trace))
                   (not (committed-p (car trace))))
              t
              (trace-exist-uncommitted-LSRM-before-p i idx (cdr trace))))))

; If there is an uncommitted modifier of register idx before instruction i,
; exist-uncommitted-LRM-before-p returns t.
(defun exist-uncommitted-LRM-before-p (i idx MT)
  (declare (xargs :guard (and (INST-p i) (rname-p idx) (MAETT-p MT))))
  (trace-exist-uncommitted-LRM-before-p i idx (MT-trace MT)))

; If there is an uncommitted modifier of special register idx before
; instruction i, exist-uncommitted-LRM-before-p returns t.
(defun exist-uncommitted-LSRM-before-p (i idx MT)
  (declare (xargs :guard (and (INST-p i) (rname-p idx) (MAETT-p MT))))
  (trace-exist-uncommitted-LSRM-before-p i idx (MT-trace MT)))

(in-theory (disable exist-uncommitted-LRM-before-p
                    exist-uncommitted-LSRM-before-p))

(defun trace-LRM-before (i idx trace)
  (declare (xargs :guard (and (INST-p i) (rname-p idx) (INST-listp trace))))
  (if (endp trace)
      nil
      (if (equal (car trace) i)
          nil
          (if (and (reg-modifier-p idx (car trace))
                   (not (trace-exist-LRM-before-p i idx (cdr trace))))
              (car trace)
              (trace-LRM-before i idx (cdr trace))))))

(defun trace-LSRM-before (i idx trace)
  (declare (xargs :guard (and (INST-p i) (rname-p idx) (INST-listp trace))))
  (if (endp trace)
      nil
      (if (equal (car trace) i)
          nil
          (if (and (sreg-modifier-p idx (car trace))
                   (not (trace-exist-LSRM-before-p i idx (cdr trace))))
```

```
             (car trace)
             (trace-LSRM-before i idx (cdr trace))))))

; If there is a register modifier before instruction i,
; LRM-before returns the last register modifier.
(defun LRM-before (i idx MT)
  (declare (xargs :guard (and (INST-p i) (rname-p idx) (MAETT-p MT))))
  (trace-LRM-before i idx (MT-trace MT)))

; If there is a special register modifier before instruction i,
; LSRM-before returns the last special register modifier.
(defun LSRM-before (i idx MT)
  (declare (xargs :guard (and (INST-p i) (rname-p idx) (MAETT-p MT))))
  (trace-LSRM-before i idx (MT-trace MT)))

(defthm inst-p-trace-LRM-before
    (implies (and (INST-p i)
                  (INST-listp trace)
                  (trace-exist-LRM-before-p i rname trace))
             (INST-p (trace-LRM-before i rname trace))))

; LRM-before returns INST-p if there exists a register modifier.
(defthm inst-p-LRM-before
    (implies (and (INST-p i) (MAETT-p MT)
                  (exist-LRM-before-p i rname MT))
             (INST-p (LRM-before i rname MT)))
  :hints (("goal" :in-theory (enable exist-LRM-before-p
                                     LRM-before))))

(defthm inst-p-trace-LSRM-before-help
    (implies (and (INST-p i) (INST-listp trace)
                  (trace-exist-LSRM-before-p i rname trace))
             (INST-p (trace-LSRM-before i rname trace))))

; LSRM-before returns INST-p if there exists a
; special register modifier.
(defthm inst-p-LSRM-before
    (implies (and (INST-p i) (MAETT-p MT)
                  (exist-LSRM-before-p i rname MT))
             (INST-p (LSRM-before i rname MT)))
  :hints (("goal" :in-theory (enable exist-LSRM-before-p
                                     LSRM-before))))

(in-theory (disable LRM-before
                    LSRM-before))

(defthm INST-p-trace-LRM-in-ROB
    (implies (and (rname-p idx) (INST-listp trace)
                  (trace-exist-LRM-in-ROB-p idx trace))
             (INST-p (trace-LRM-in-ROB idx trace))))

; LRM-in-ROB returns an INST-p if there exist register
; modifiers in the ROB.
(defthm INST-p-LRM-in-ROB
    (implies (and (rname-p idx) (MAETT-p MT)
                  (exist-LRM-in-ROB-p idx MT))
             (INST-p (LRM-in-ROB idx MT)))
  :hints (("goal" :in-theory (enable exist-LRM-in-ROB-p
                                     LRM-in-ROB))))

(defthm INST-p-trace-LSRM-in-ROB
    (implies (and (srname-p idx) (INST-listp trace)
```

```
                (trace-exist-LSRM-in-ROB-p idx trace))
           (INST-p (trace-LSRM-in-ROB idx trace))))

; LSRM-in-ROB returns an INST-p if there exist special
; register modifiers in the ROB.
(defthm INST-p-LSRM-in-ROB
    (implies (and (srname-p idx) (MAETT-p MT)
                (exist-LSRM-in-ROB-p idx MT))
           (INST-p (LSRM-in-ROB idx MT)))
  :hints (("goal" :in-theory (enable exist-LSRM-in-ROB-p
                                     LSRM-in-ROB))))

(defthm trace-exist-LRM-if-trace-exist-uncommitted-LRM
    (implies (trace-exist-uncommitted-LRM-before-p i r trace)
           (trace-exist-LRM-before-p i r trace))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
     (implies (not (trace-exist-LRM-before-p i r trace))
              (not (trace-exist-uncommitted-LRM-before-p i r trace))))))

; The existence of uncommitted register modifier implies
; the existence of register modifiers in general.
(defthm exist-LRM-if-exist-uncommitted-LRM
    (implies (exist-uncommitted-LRM-before-p i r MT)
           (exist-LRM-before-p i r MT))
  :hints (("goal" :in-theory (enable exist-uncommitted-LRM-before-p
                                     exist-LRM-before-p))))

(defthm trace-exist-LSRM-if-exist-uncommitted-LSRM
    (implies (trace-exist-uncommitted-LSRM-before-p i r trace)
           (trace-exist-LSRM-before-p i r trace))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
     (implies (not (trace-exist-LSRM-before-p i r trace))
              (not (trace-exist-uncommitted-LSRM-before-p i r trace))))))

; The existence of uncommitted special register modifier implies
; the existence of special register modifiers in general.
(defthm exist-LSRM-if-exist-uncommitted-LSRM
    (implies (exist-uncommitted-LSRM-before-p i r MT)
           (exist-LSRM-before-p i r MT))
  :hints (("goal" :in-theory (enable exist-uncommitted-LSRM-before-p
                                     exist-LSRM-before-p))))

;;;;;;;;;;;;;;;;;;;Definition of mem-modifier;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Definition of memory modifiers. We define the same concept as
; register modifier with the memory.  We define
; (LMM-before i addr MT) as the last instruction that
; modifies the memory at address addr before the instruction i.
;
; We didn't define a concept like LRM-in-ROB.  This
; was useful in obtaining the ideal value for the field of the
; reservation station which designates the producer of the source
; operand value.   Our load store unit does load-bypassing, but this
; is done by comparing access address directly.  Thus we need not to
; generate something like Tomasulo's tag.  Thus, we did not find the
; use of functions similar to LRM-in-ROB.
(defun mem-modifier-p (addr i)
  (declare (xargs :guard (and (addr-p addr) (INST-p i))))
  (and (b1p (INST-store? i))
```

```
      (equal (INST-store-addr i) addr)))
(in-theory (disable mem-modifier-p))

(defun trace-exist-LMM-before-p (i addr trace)
  (declare (xargs :guard (and (INST-p i) (addr-p addr)
                              (INST-listp trace))))
  (if (endp trace)
      nil
      (if (equal (car trace) i)
          nil
          (if (mem-modifier-p addr (car trace))
              t
              (trace-exist-LMM-before-p i addr (cdr trace))))))

; If there is a memory modifier before instruction i,
; exist-LMM-before-p returns t.
(defun exist-LMM-before-p (i addr MT)
  (declare (xargs :guard (and (INST-p i) (addr-p addr) (MAETT-p MT))))
  (trace-exist-LMM-before-p i addr (MT-trace MT)))

(in-theory (disable exist-LMM-before-p))

(defun trace-LMM-before (i addr trace)
  (declare (xargs :guard (and (INST-p i) (addr-p addr) (INST-listp trace))))
  (if (endp trace)
      nil
      (if (equal (car trace) i)
          nil
          (if (and (mem-modifier-p addr (car trace))
                   (not (trace-exist-LMM-before-p i addr (cdr trace))))
              (car trace)
              (trace-LMM-before i addr (cdr trace))))))

; If there is an memory modifier before instruction i,
; LMM-before returns the last memory modifier.
(defun LMM-before (i addr MT)
  (declare (xargs :guard (and (INST-p i) (addr-p addr) (MAETT-p MT))))
  (trace-LMM-before i addr (MT-trace MT)))


(defun trace-exist-non-retired-LMM-before-p (i addr trace)
  (declare (xargs :guard (and (INST-p i) (addr-p addr)
                              (INST-listp trace))))
  (if (endp trace)
      nil
      (if (equal (car trace) i)
          nil
          (if (and (mem-modifier-p addr (car trace))
                   (not (retire-stg-p (INST-stg (car trace)))))
              t
              (trace-exist-non-retired-LMM-before-p i addr (cdr trace))))))

; If there is a non-retired memory modifier before instruction i,
; exist-non-retired-LMM-before-p returns non nil.
(defun exist-non-retired-LMM-before-p (i addr MT)
  (declare (xargs :guard (and (INST-p i) (addr-p addr) (MAETT-p MT))))
  (trace-exist-non-retired-LMM-before-p i addr (MT-trace MT)))

(in-theory (disable exist-non-retired-LMM-before-p))

(encapsulate nil
(local
```

```
 (defthm inst-p-LMM-before-help
     (implies (and (INST-p i)
                   (INST-listp trace)
                   (trace-exist-LMM-before-p i addr trace))
              (INST-p (trace-LMM-before i addr trace)))))

; LMM-before returns INST-p if there is a memory modifier.
(defthm inst-p-LMM-before
    (implies (and (INST-p i) (MAETT-p MT)
                  (exist-LMM-before-p i addr MT))
             (INST-p (LMM-before i addr MT)))
  :hints (("goal" :in-theory (enable exist-LMM-before-p
                                     LMM-before))))
)
(in-theory (disable LMM-before))

; Consistent-reg-ref-p defines the correct state of register reference
; table at entry idx.  If wait? flag in the register reference table
; is on for register idx, then there is a register modifier in ROB,
; and its tag is stored in the register reference table.
; If wait? flag is not on, the register modifier does not exist.
; Note that the register reference table may not contain the correct
; value when the processor is executing speculatively or modified
; instruction stream at the dispatching boundary.
(defun consistent-reg-ref-p (idx MT MA)
  (declare (xargs :guard (and (rname-p idx) (MAETT-p MT) (MA-state-p MA))))
  (if (or (b1p (MT-specultv-at-dispatch? MT))
          (b1p (MT-modified-at-dispatch? MT)))
      t
      (if (b1p (reg-ref-wait? (reg-tbl-nth idx (DQ-reg-tbl (MA-DQ MA)))))
          (and (exist-LRM-in-ROB-p idx MT)
               (equal (INST-tag (LRM-in-ROB idx MT))
                      (reg-ref-tag (reg-tbl-nth idx (DQ-reg-tbl (MA-DQ MA))))))
          (not (exist-LRM-in-ROB-p idx MT)))))


; Similar to consistent-reg-ref-p.  It specifies that the register
; reference table for special registers contain the correct values.
(defun consistent-sreg-ref-p (idx MT MA)
  (declare (xargs :guard (and (srname-p idx) (MAETT-p MT) (MA-state-p MA))
                  :guard-hints (("goal" :in-theory (enable srname-p)))))
  (if (or (b1p (MT-specultv-at-dispatch? MT))
          (b1p (MT-modified-at-dispatch? MT)))
      T
    (if (b1p (reg-ref-wait? (sreg-tbl-nth idx (DQ-sreg-tbl (MA-DQ MA)))))
        (and (exist-LSRM-in-ROB-p idx MT)
             (equal (INST-tag (LSRM-in-ROB idx MT))
                    (reg-ref-tag (sreg-tbl-nth idx (DQ-sreg-tbl (MA-DQ MA))))))
        (not (exist-LSRM-in-ROB-p idx MT)))))

(defun consistent-reg-tbl-under (idx MT MA)
  (declare (xargs :guard (and (integerp idx) (<= 0 idx) (<= idx *num-regs*)
                              (MAETT-p MT) (MA-state-p MA))
                  :guard-hints
                  (("goal" :in-theory (enable rname-p unsigned-byte-p)))))
  (if (zp idx)
      t
      (and (consistent-reg-ref-p (1- idx) MT MA)
           (consistent-reg-tbl-under (1- idx) MT MA))))

; All entries in the register reference table contain the right
; values.
```

```
(defun consistent-reg-tbl-p (MT MA)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA))))
  (consistent-reg-tbl-under *num-regs* MT MA))

(defun consistent-sreg-tbl-p (MT MA)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA))))
  (and (consistent-sreg-ref-p 0 MT MA)
       (consistent-sreg-ref-p 1 MT MA)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Definition of consistent-RS-p
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Consistent-RS-p defines the right values for the reservation
; stations.  These conditions are critical to guarantee that the
; source operand values are obtained for the right source.
;
; For example, let us consider consistent-IU-RS0-p.  All other
; predicates are straightforward once we understand consistent-IU-RS0-p.
; Consistent-IU-RS0-p is a conjunct of several clauses.  A hypothesis
; (not (b1p (RS-ready1? RS0))) of the first clause checks whether the
; first operand for instruction i is not ready.
; (b1p (logbit 0 (cntlv-operand cntlv))) indicates the operand
; type for which source operands are general registers RA and RB.
; In this case, the first clause claims three things.  First, a
; uncommitted modifier of register RA exists before instruction i.
; Second, the last register modifier of RA is still in execution
; stage. (If it is in the complete stage, the reservation station
; should have already obtained its value.)  Third, the src1 field of
; reservation station R0 contains the tag to the last register
; modifier.  If instruction i is speculatively executed, or in the
; modified stream of instruction, invariants always hold.
(deflabel begin-consistent-RS-p-def)
(defun consistent-IU-RS0-p (i MT MA)
  (declare (xargs :guard (and (INST-p i) (MAETT-p MT) (MA-state-p MA))))
  (let ((RS0 (IU-RS0 (MA-IU MA)))
        (RA (INST-RA i))
        (RB (INST-RB i))
        (RC (INST-RC i))
        (cntlv (INST-cntlv i)))
    (and (implies (and (not (b1p (inst-specultv? i)))
                       (not (b1p (INST-modified? i)))
                       (not (b1p (RS-ready1? RS0)))
                       (b1p (logbit 0 (cntlv-operand cntlv))))
                  (and (exist-uncommitted-LRM-before-p i RA MT)
                       (execute-stg-p
                        (INST-stg (LRM-before i RA MT)))
                       (equal (RS-src1 RS0)
                              (INST-tag (LRM-before i RA MT)))))
         (implies (and (not (b1p (inst-specultv? i)))
                       (not (b1p (INST-modified? i)))
                       (not (b1p (RS-ready2? RS0)))
                       (b1p (logbit 0 (cntlv-operand cntlv))))
                  (and (exist-uncommitted-LRM-before-p i RB MT)
                       (execute-stg-p
                        (INST-stg (LRM-before i RB MT)))
                       (equal (RS-src2 RS0)
                              (INST-tag (LRM-before i RB MT)))))
         (implies (and (not (b1p (inst-specultv? i)))
                       (not (b1p (INST-modified? i)))
                       (not (b1p (RS-ready1? RS0)))
                       (b1p (logbit 2 (cntlv-operand cntlv))))
                  (and (exist-uncommitted-LRM-before-p i RC MT)
```

```
                        (execute-stg-p
                         (INST-stg (LRM-before i RC MT)))
                        (equal (RS-src1 RS0)
                               (INST-tag (LRM-before i RC MT)))))
          (implies (and (not (b1p (inst-specultv? i)))
                        (not (b1p (INST-modified? i)))
                        (not (b1p (RS-ready1? RS0)))
                        (b1p (logbit 3 (cntlv-operand cntlv))))
                   (and (exist-uncommitted-LSRM-before-p i RA MT)
                        (execute-stg-p
                         (INST-stg (LSRM-before i RA MT)))
                        (equal (RS-src1 RS0)
                               (INST-tag (LSRM-before i RA MT))))))))

(defun consistent-IU-RS1-p (i MT MA)
  (declare (xargs :guard (and (INST-p i) (MAETT-p MT) (MA-state-p MA))))
  (let ((RS1 (IU-RS1 (MA-IU MA)))
        (RA (INST-RA i))
        (RB (INST-RB i))
        (RC (INST-RC i))
        (cntlv (INST-cntlv i)))
    (and (implies (and (not (b1p (inst-specultv? i)))
                       (not (b1p (INST-modified? i)))
                       (not (b1p (RS-ready1? RS1)))
                       (b1p (logbit 0 (cntlv-operand cntlv))))
                  (and (exist-uncommitted-LRM-before-p i RA MT)
                       (execute-stg-p
                        (INST-stg (LRM-before i RA MT)))
                       (equal (RS-src1 RS1)
                              (INST-tag (LRM-before i RA MT)))))
         (implies (and (not (b1p (inst-specultv? i)))
                       (not (b1p (INST-modified? i)))
                       (not (b1p (RS-ready2? RS1)))
                       (b1p (logbit 0 (cntlv-operand cntlv))))
                  (and (exist-uncommitted-LRM-before-p i RB MT)
                       (execute-stg-p
                        (INST-stg (LRM-before i RB MT)))
                       (equal (RS-src2 RS1)
                              (INST-tag (LRM-before i RB MT)))))
         (implies (and (not (b1p (inst-specultv? i)))
                       (not (b1p (INST-modified? i)))
                       (not (b1p (RS-ready1? RS1)))
                       (b1p (logbit 2 (cntlv-operand cntlv))))
                  (and (exist-uncommitted-LRM-before-p i RC MT)
                       (execute-stg-p
                        (INST-stg (LRM-before i RC MT)))
                       (equal (RS-src1 RS1)
                              (INST-tag (LRM-before i RC MT)))))
         (implies (and (not (b1p (inst-specultv? i)))
                       (not (b1p (INST-modified? i)))
                       (not (b1p (RS-ready1? RS1)))
                       (b1p (logbit 3 (cntlv-operand cntlv))))
                  (and (exist-uncommitted-LSRM-before-p i RA MT)
                       (execute-stg-p
                        (INST-stg (LSRM-before i RA MT)))
                       (equal (RS-src1 RS1)
                              (INST-tag (LSRM-before i RA MT))))))))

(defun consistent-IU-RS-p (i MT MA)
  (declare (xargs :guard (and (INST-p i) (MAETT-p MT) (MA-state-p MA))))
  (cond ((equal (INST-stg i) '(IU RS0))
         (consistent-IU-RS0-p i MT MA))
```

```
        ((equal (INST-stg i) '(IU RS1))
         (consistent-IU-RS1-p i MT MA))
        (t t)))

(defun consistent-MU-RS0-p (i MT MA)
  (declare (xargs :guard (and (INST-p i) (MAETT-p MT) (MA-state-p MA))))
  (let ((RS0 (MU-RS0 (MA-MU MA)))
        (RA (INST-RA i))
        (RB (INST-RB i)))
    (and (implies (and (not (b1p (inst-specultv? i)))
                       (not (b1p (INST-modified? i)))
                       (not (b1p (RS-ready1? RS0))))
                  (and (exist-uncommitted-LRM-before-p i RA MT)
                       (execute-stg-p
                        (INST-stg (LRM-before i RA MT)))
                       (equal (RS-src1 RS0)
                              (INST-tag (LRM-before i RA MT)))))
         (implies (and (not (b1p (inst-specultv? i)))
                       (not (b1p (INST-modified? i)))
                       (not (b1p (RS-ready2? RS0))))
                  (and (exist-uncommitted-LRM-before-p i RB MT)
                       (execute-stg-p
                        (INST-stg (LRM-before i RB MT)))
                       (equal (RS-src2 RS0)
                              (INST-tag (LRM-before i RB MT))))))))

(defun consistent-MU-RS1-p (i MT MA)
  (declare (xargs :guard (and (INST-p i) (MAETT-p MT) (MA-state-p MA))))
  (let ((RS1 (MU-RS1 (MA-MU MA)))
        (RA (INST-RA i))
        (RB (INST-RB i)))
    (and (implies (and (not (b1p (inst-specultv? i)))
                       (not (b1p (INST-modified? i)))
                       (not (b1p (RS-ready1? RS1))))
                  (and (exist-uncommitted-LRM-before-p i RA MT)
                       (execute-stg-p
                        (INST-stg (LRM-before i RA MT)))
                       (equal (RS-src1 RS1)
                              (INST-tag (LRM-before i RA MT)))))
         (implies (and (not (b1p (inst-specultv? i)))
                       (not (b1p (INST-modified? i)))
                       (not (b1p (RS-ready2? RS1))))
                  (and (exist-uncommitted-LRM-before-p i RB MT)
                       (execute-stg-p
                        (INST-stg (LRM-before i RB MT)))
                       (equal (RS-src2 RS1)
                              (INST-tag (LRM-before i RB MT))))))))

(defun consistent-MU-RS-p (i MT MA)
  (declare (xargs :guard (and (INST-p i) (MAETT-p MT) (MA-state-p MA))))
  (cond ((equal (INST-stg i) '(MU RS0))
         (consistent-MU-RS0-p i MT MA))
        ((equal (INST-stg i) '(MU RS1))
         (consistent-MU-RS1-p i MT MA))
        (t t)))

(defun consistent-BU-RS0-p (i MT MA)
  (declare (xargs :guard (and (INST-p i) (MAETT-p MT) (MA-state-p MA))))
  (let ((RS0 (BU-RS0 (MA-BU MA)))
        (RC (INST-RC i)))
    (and (implies (and (not (b1p (inst-specultv? i)))
                       (not (b1p (INST-modified? i)))
```

```
                    (not (b1p (BU-RS-ready? RS0))))
               (and (exist-uncommitted-LRM-before-p i RC MT)
                    (execute-stg-p
                     (INST-stg (LRM-before i RC MT)))
                    (equal (BU-RS-src RS0)
                           (INST-tag (LRM-before i RC MT))))))))

(defun consistent-BU-RS1-p (i MT MA)
  (declare (xargs :guard (and (INST-p i) (MAETT-p MT) (MA-state-p MA))))
  (let ((RS1 (BU-RS1 (MA-BU MA)))
        (RC (INST-RC i)))
    (and (implies (and (not (b1p (inst-specultv? i)))
                       (not (b1p (INST-modified? i)))
                       (not (b1p (BU-RS-ready? RS1))))
                  (and (exist-uncommitted-LRM-before-p i RC MT)
                       (execute-stg-p
                        (INST-stg (LRM-before i RC MT)))
                       (equal (BU-RS-src RS1)
                              (INST-tag (LRM-before i RC MT))))))))

(defun consistent-BU-RS-p (i MT MA)
  (declare (xargs :guard (and (INST-p i) (MAETT-p MT) (MA-state-p MA))))
  (cond ((equal (INST-stg i) '(BU RS0))
         (consistent-BU-RS0-p i MT MA))
        ((equal (INST-stg i) '(BU RS1))
         (consistent-BU-RS1-p i MT MA))
        (t t)))

(defun consistent-LSU-RS0-p (i MT MA)
  (declare (xargs :guard (and (INST-p i) (MAETT-p MT) (MA-state-p MA))))
  (let ((RS0 (LSU-RS0 (MA-LSU MA)))
        (RA (INST-RA i))
        (RB (INST-RB i))
        (RC (INST-RC i)))
    (and (implies (and (not (b1p (inst-specultv? i)))
                       (not (b1p (INST-modified? i)))
                       (not (b1p (LSU-RS-rdy1? RS0))))
                  (and (exist-uncommitted-LRM-before-p i RA MT)
                       (execute-stg-p
                        (INST-stg (LRM-before i RA MT)))
                       (equal (LSU-RS-src1 RS0)
                              (INST-tag (LRM-before i RA MT)))))
         (implies (and (not (b1p (inst-specultv? i)))
                       (not (b1p (INST-modified? i)))
                       (not (b1p (LSU-RS-rdy2? RS0))))
                  (and (exist-uncommitted-LRM-before-p i RB MT)
                       (execute-stg-p
                        (INST-stg (LRM-before i RB MT)))
                       (equal (LSU-RS-src2 RS0)
                              (INST-tag (LRM-before i RB MT)))))
         (implies (and (not (b1p (inst-specultv? i)))
                       (not (b1p (INST-modified? i)))
                       (not (b1p (LSU-RS-rdy3? RS0))))
                  (and (exist-uncommitted-LRM-before-p i RC MT)
                       (execute-stg-p
                        (INST-stg (LRM-before i RC MT)))
                       (equal (LSU-RS-src3 RS0)
                              (INST-tag (LRM-before i RC MT))))))))

(defun consistent-LSU-RS1-p (i MT MA)
  (declare (xargs :guard (and (INST-p i) (MAETT-p MT) (MA-state-p MA))))
  (let ((RS1 (LSU-RS1 (MA-LSU MA)))
```

```
        (RA (INST-RA i))
        (RB (INST-RB i))
        (RC (INST-RC i)))
    (and (implies (and (not (b1p (inst-specultv? i)))
                       (not (b1p (INST-modified? i)))
                       (not (b1p (LSU-RS-rdy1? RS1))))
                  (and (exist-uncommitted-LRM-before-p i RA MT)
                       (execute-stg-p
                        (INST-stg (LRM-before i RA MT)))
                       (equal (LSU-RS-src1 RS1)
                              (INST-tag (LRM-before i RA MT)))))
         (implies (and (not (b1p (inst-specultv? i)))
                       (not (b1p (INST-modified? i)))
                       (not (b1p (LSU-RS-rdy2? RS1))))
                  (and (exist-uncommitted-LRM-before-p i RB MT)
                       (execute-stg-p
                        (INST-stg (LRM-before i RB MT)))
                       (equal (LSU-RS-src2 RS1)
                              (INST-tag (LRM-before i RB MT)))))
         (implies (and (not (b1p (inst-specultv? i)))
                       (not (b1p (INST-modified? i)))
                       (not (b1p (LSU-RS-rdy3? RS1))))
                  (and (exist-uncommitted-LRM-before-p i RC MT)
                       (execute-stg-p
                        (INST-stg (LRM-before i RC MT)))
                       (equal (LSU-RS-src3 (LSU-RS1 (MA-LSU MA)))
                              (INST-tag (LRM-before i RC MT)))))))))

(defun consistent-LSU-RS-p (i MT MA)
  (declare (xargs :guard (and (INST-p i) (MAETT-p MT) (MA-state-p MA))))
  (cond ((equal (INST-stg i) '(LSU RS0))
         (consistent-LSU-RS0-p i MT MA))
        ((equal (INST-stg i) '(LSU RS1))
         (consistent-LSU-RS1-p i MT MA))
        (t t)))

; Consistent-RS-entry-p is true if the reservation station stores
; correct info about i.
(defun consistent-RS-entry-p (i MT MA)
  (declare (xargs :guard (and (INST-p i) (MAETT-p MT) (MA-state-p MA))))
  (cond ((IU-stg-p (INST-stg i)) (consistent-IU-RS-p i MT MA))
        ((MU-stg-p (INST-stg i)) (consistent-MU-RS-p i MT MA))
        ((BU-stg-p (INST-stg i)) (consistent-BU-RS-p i MT MA))
        ((LSU-stg-p (INST-stg i)) (consistent-LSU-RS-p i MT MA))
        (t t)))

(defun trace-consistent-RS-p (trace MT MA)
  (declare (xargs :guard (and (INST-listp trace)
                              (MAETT-p MT) (MA-state-p MA))))
  (if (endp trace)
      t
      (and (consistent-RS-entry-p (car trace) MT MA)
           (trace-consistent-RS-p (cdr trace) MT MA))))

; All reservation stations contain the right values to keep track of
; instructions that supply the operand values.
(defun consistent-RS-p (MT MA)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA))))
  (trace-consistent-RS-p (MT-trace MT) MT MA))
(deflabel end-consistent-RS-p-def)
(deftheory consistent-RS-p-def
    (set-difference-theories (universal-theory 'end-consistent-RS-p-def)
```

```
                              (universal-theory 'begin-consistent-RS-p-def)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Definition of in-order-ROB-p
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ROB is a FIFO buffer.  Instructions are dispatched and committed in
; program order, and the ROB records them in that order.  Since the
; pointer to the head is incremented every time a new entry comes in,
; and wrapped around when the bottom is reached.  Thus, the ROB
; index idx1 to instruction I1 and index idx2 to the next instruction
; I2 are related as idx2 = (rob-index (+ 1 idx1)).  See MA2-def.lisp
; for detail.
(defun in-order-ROB-trace-p (trace idx)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace)
      T
      (if (or (IFU-stg-p (INST-stg (car trace)))
              (DQ-stg-p (INST-stg (car trace))))
          T
          (if (or (complete-stg-p (INST-stg (car trace)))
                  (execute-stg-p (INST-stg (car trace))))
              (and (equal (INST-tag (car trace)) idx)
                   (in-order-ROB-trace-p (cdr trace) (rob-index (+ 1 idx))))
              (in-order-ROB-trace-p (cdr trace) idx)))))

(defun in-order-ROB-p (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (in-order-ROB-trace-p (MT-trace MT) (MT-rob-head MT)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Definition of no-stage-conflict
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; No-stage-conflict defines that no structural conflicts occur on
; pipeline latches.
; No two instructions share the same stage.  This is verified by case
; analysis on each pipeline stage.  For instance, no-IFU-stg-conflict
; guarantees that no instruction share the IFU stage at a time.
; This is represented by uniq-INST-at-stg and no-INST-at-stg.
; If IFU-valid? is 1, there exists only one instruction at stage
; (IFU).  If 0, no instruction should be at (IFU).
(defun no-IFU-stg-conflict (MT MA)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA))))
  (if (b1p (IFU-valid? (MA-IFU MA)))
      (uniq-INST-at-stg '(IFU) MT)
      (no-INST-at-stg '(IFU) MT)))

(defun no-DQ-stg-conflict (MT MA)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA))))
  (and (if (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))
           (uniq-INST-at-stg '(DQ 0) MT)
           (no-INST-at-stg '(DQ 0) MT))
       (if (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))
           (uniq-INST-at-stg '(DQ 1) MT)
           (no-INST-at-stg '(DQ 1) MT))
       (if (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))
           (uniq-INST-at-stg '(DQ 2) MT)
           (no-INST-at-stg '(DQ 2) MT))
       (if (b1p (DE-valid? (DQ-DE3 (MA-DQ MA))))
           (uniq-INST-at-stg '(DQ 3) MT)
           (no-INST-at-stg '(DQ 3) MT))))

(defun no-IU-stg-conflict (MT MA)
```

```
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA))))
  (and (if (b1p (RS-valid? (IU-RS0 (MA-IU MA))))
           (uniq-INST-at-stg '(IU RS0) MT)
           (no-INST-at-stg '(IU RS0) MT))
       (if (b1p (RS-valid? (IU-RS1 (MA-IU MA))))
           (uniq-INST-at-stg '(IU RS1) MT)
           (no-INST-at-stg '(IU RS1) MT))))

(defun no-BU-stg-conflict (MT MA)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA))))
  (and (if (b1p (BU-RS-valid? (BU-RS0 (MA-BU MA))))
           (uniq-INST-at-stg '(BU RS0) MT)
           (no-INST-at-stg '(BU RS0) MT))
       (if (b1p (BU-RS-valid? (BU-RS1 (MA-BU MA))))
           (uniq-INST-at-stg '(BU RS1) MT)
           (no-INST-at-stg '(BU RS1) MT))))

(defun no-MU-stg-conflict (MT MA)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA))))
  (and (if (b1p (RS-valid? (MU-RS0 (MA-MU MA))))
           (uniq-INST-at-stg '(MU RS0) MT)
           (no-INST-at-stg '(MU RS0) MT))
       (if (b1p (RS-valid? (MU-RS1 (MA-MU MA))))
           (uniq-INST-at-stg '(MU RS1) MT)
           (no-INST-at-stg '(MU RS1) MT))
       (if (b1p (MU-latch1-valid? (MU-lch1 (MA-MU MA))))
           (uniq-INST-at-stg '(MU lch1) MT)
           (no-INST-at-stg '(MU lch1) MT))
       (if (b1p (MU-latch2-valid? (MU-lch2 (MA-MU MA))))
           (uniq-INST-at-stg '(MU lch2) MT)
           (no-INST-at-stg '(MU lch2) MT))))

(defun no-LSU-stg-conflict (MT MA)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA))))
  (and (if (b1p (LSU-RS-valid? (LSU-RS0 (MA-LSU MA))))
           (uniq-INST-at-stg '(LSU RS0) MT)
           (no-INST-at-stg '(LSU RS0) MT))
       (if (b1p (LSU-RS-valid? (LSU-RS1 (MA-LSU MA))))
           (uniq-INST-at-stg '(LSU RS1) MT)
           (no-INST-at-stg '(LSU RS1) MT))
       (if (b1p (rbuf-valid? (LSU-rbuf (MA-LSU MA))))
           (uniq-INST-at-stg '(LSU rbuf) MT)
           (no-INST-at-stg '(LSU rbuf) MT))
       (if (b1p (LSU-latch-valid? (LSU-lch (MA-LSU MA))))
           (uniq-INST-at-stgs '((LSU lch)
                                (LSU wbuf0 lch)
                                (LSU wbuf1 lch))
                              MT)
           (no-INST-at-stgs '((LSU lch)
                              (LSU wbuf0 lch)
                              (LSU wbuf1 lch))
                            MT))
       (if (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
           (uniq-INST-at-stgs '((LSU wbuf0)
                                (LSU wbuf0 lch)
                                (complete wbuf0)
                                (commit wbuf0))
                              MT)
           (no-INST-at-stgs '((LSU wbuf0)
                              (LSU wbuf0 lch)
                              (complete wbuf0)
                              (commit wbuf0))
```

```
                          MT))
       (if (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
           (uniq-INST-at-stgs '((LSU wbuf1)
                                (LSU wbuf1 lch)
                                (complete wbuf1)
                                (commit wbuf1))
                              MT)
           (no-INST-at-stgs '((LSU wbuf1)
                              (LSU wbuf1 lch)
                              (complete wbuf1)
                              (commit wbuf1))
                            MT))))

; No structural conflicts occur at any pipeline latch.  See comments above.
(defun no-stage-conflict (MT MA)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA))))
  (and (no-IFU-stg-conflict MT MA)
       (no-DQ-stg-conflict MT MA)
       (no-IU-stg-conflict MT MA)
       (no-MU-stg-conflict MT MA)
       (no-LSU-stg-conflict MT MA)
       (no-BU-stg-conflict MT MA)))

; No-tag-conflict defines that no structural conflict occurs in the
; ROB.  In other words, no ROB entry is shared by more than one
; instruction.  It also implies that the tag used in the Tomasulo's
; algorithm is unique, because we use the index to the ROB entry which
; is allocated to an instruction as its tag.
;
; No-tag-conflict-at defines whether there is no structural hazard at
; ROB entry indexed by idx.  If valid? flag of that entry is 1, there
; exists a uniq instruction assigned to that entry. If 0, no
; instruction is assigned.
(defun no-tag-conflict-at (idx MT MA)
  (declare (xargs :guard (and (rob-index-p idx) (MAETT-p MT) (MA-state-p MA))))
  (if (b1p (robe-valid? (nth-robe idx (MA-rob MA))))
      (uniq-inst-of-tag idx MT)
      (no-inst-of-tag idx MT)))

(defun no-tag-conflict-under (idx MT MA)
  (declare (xargs :guard (and (integerp idx) (<= 0 idx) (<= idx *rob-size*)
                              (MAETT-p MT) (MA-state-p MA))
                  :guard-hints
                  (("goal" :in-theory (enable rob-index-p unsigned-byte-p)))))
  (if (zp idx)
      t
      (and (no-tag-conflict-at (1- idx) MT MA)
           (no-tag-conflict-under (1- idx) MT MA))))

; No structural conflict occurs in the ROB.
; See the comment above.
(defun no-tag-conflict (MT MA)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA))))
  (no-tag-conflict-under *rob-size* MT MA))

;;;; Definition of misc-inv
; correct-entries-in-DQ-p is a part of misc-inv.
; This function checks whether the dispatch queue contains the correct
; number of instructions suggested by MT-DQ-len.  If index is smaller
; than MT-DQ-len, the entry with the index should contain an instruction.
(defun correct-entries-in-DQ-p (MT MA)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA))))
```

```
  (and (iff (< 0 (MT-DQ-len MT)) (b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))
       (iff (< 1 (MT-DQ-len MT)) (b1p (DE-valid? (DQ-DE1 (MA-DQ MA)))))
       (iff (< 2 (MT-DQ-len MT)) (b1p (DE-valid? (DQ-DE2 (MA-DQ MA)))))
       (iff (< 3 (MT-DQ-len MT)) (b1p (DE-valid? (DQ-DE3 (MA-DQ MA)))))))

; Misc invariants.
; MT-rob-head and MT-rob-tail records the correct index of head and tail
; entries in ROB.
; MT-DQ-len must record the number of instructions in dispatch queue.
;
(defun misc-inv (MT MA)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA))))
  (and (equal (rob-flg (MA-rob MA)) (MT-rob-flg MT))
       (equal (rob-head (MA-rob MA)) (MT-rob-head MT))
       (equal (rob-tail (MA-rob MA)) (MT-rob-tail MT))
       (<= (MT-DQ-len MT) 4)
       (correct-entries-in-DQ-p MT MA)))

; Definition of strong invariants, which is a conjunction of all properties
; defined above.
(defun inv (MT MA)
  (declare (xargs :guard (and (MAETT-p MT) (MA-state-p MA))))
  (and (weak-inv MT)
       (pc-match-p MT MA)
       (SRF-match-p MT MA)
       (RF-match-p MT MA)
       (mem-match-p MT MA)
       (no-specultv-commit-p MT)
       (correct-speculation-p MT)
       (correct-exintr-p MT)
       (MT-INST-inv MT MA)
       (in-order-dispatch-commit-p MT)
       (in-order-DQ-p MT)
       (in-order-LSU-inst-p MT MA)
       (in-order-ROB-p MT)
       (consistent-RS-p MT MA)
       (consistent-reg-tbl-p MT MA)
       (consistent-sreg-tbl-p MT MA)
       (no-stage-conflict MT MA)
       (no-tag-conflict MT MA)
       (consistent-MA-p MA)
       (misc-inv MT MA)))

(deflabel end-invariants-def)

(deftheory invariants-def
    (set-difference-theories (function-theory 'end-invariants-def)
                             (function-theory 'begin-invariants-def)))

(deftheory invariants-def-non-rec-functions
    (non-rec-functions (theory 'invariants-def) world))

(in-theory (disable invariants-def-non-rec-functions))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Forward-chaining of weak invariants wk-inv
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deflabel begin-invariants-forward-lemmas)

(defthm weak-invariants-forward
    (implies (inv MT MA)
             (weak-inv MT))
```

```
  :hints (("goal" :In-theory (enable inv)))
  :rule-classes :forward-chaining)

(defthm MT-new-ID-distinct-p-forward-2
    (implies (weak-inv MT)
             (MT-new-ID-distinct-p MT))
  :hints (("Goal" :in-theory (enable weak-inv)))
  :rule-classes :forward-chaining)

(defthm MT-distinct-IDs-p-forward-2
    (implies (weak-inv MT)
             (MT-distinct-IDs-p MT))
  :hints (("Goal" :in-theory (enable weak-inv)))
  :rule-classes :forward-chaining)

(defthm MT-distinct-chain-p-forward-2
    (implies (weak-inv MT)
             (MT-distinct-INST-p MT))
  :hints (("Goal" :in-theory (enable weak-inv)))
  :rule-classes :forward-chaining)

(defthm ISA-step-chain-p-forward-2
    (implies (weak-inv MT)
             (ISA-step-chain-p MT))
  :hints (("Goal" :in-theory (enable weak-inv)))
  :rule-classes :forward-chaining)

(defthm correct-modified-flgs-p-forward-2
    (implies (weak-inv MT)
             (correct-modified-flgs-p MT))
  :hints (("Goal" :in-theory (enable weak-inv)))
  :rule-classes :forward-chaining)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Forward-chaining for strong invariants inv
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defthm pc-match-p-forward
    (implies (inv MT MA)
             (pc-match-p MT MA))
  :hints (("goal" :in-theory (enable inv)))
  :rule-classes :forward-chaining)

(defthm SRF-match-p-forward
    (implies (inv MT MA)
             (SRF-match-p MT MA))
  :hints (("goal" :in-theory (enable inv)))
  :rule-classes :forward-chaining)

(defthm RF-match-p-forward
    (implies (inv MT MA)
             (RF-match-p MT MA))
  :hints (("goal" :in-theory (enable inv)))
  :rule-classes :forward-chaining)

(defthm mem-match-p-forward
    (implies (inv MT MA)
             (mem-match-p MT MA))
  :hints (("goal" :in-theory (enable inv)))
  :rule-classes :forward-chaining)

(defthm no-specultv-commit-p-forward
    (implies (inv MT MA)
```

```
            (no-specultv-commit-p MT))
  :hints (("goal" :in-theory (enable inv)))
  :rule-classes :forward-chaining)

(defthm correct-speculation-p-forward
    (implies (inv MT MA)
             (correct-speculation-p MT))
  :hints (("goal" :in-theory (enable inv)))
  :rule-classes :forward-chaining)

(defthm correct-exintr-p-forward
    (implies (inv MT MA)
             (correct-exintr-p MT))
  :hints (("goal" :in-theory (enable inv)))
  :rule-classes :forward-chaining)

(defthm MT-inst-inv-forward
    (implies (inv MT MA)
             (MT-INST-inv MT MA))
  :hints (("goal" :in-theory (enable inv)))
  :rule-classes :forward-chaining)

(defthm in-order-dispatch-commit-p-forward
    (implies (inv MT MA)
             (in-order-dispatch-commit-p MT))
  :hints (("goal" :in-theory (enable inv)))
  :rule-classes :forward-chaining)

(defthm in-order-DQ-p-forward
    (implies (inv MT MA)
             (in-order-DQ-p MT))
  :hints (("goal" :in-theory (enable inv)))
  :rule-classes :forward-chaining)

(defthm in-order-ROB-p-forward
    (implies (inv MT MA)
             (in-order-ROB-p MT))
  :hints (("goal" :in-theory (enable inv)))
  :rule-classes :forward-chaining)

(defthm no-stage-conflict-p-forward
    (implies (inv MT MA)
             (no-stage-conflict MT MA))
  :hints (("goal" :in-theory (enable inv)))
  :rule-classes :forward-chaining)

(defthm no-tag-conflict-p-forward
    (implies (inv MT MA)
             (no-tag-conflict MT MA))
  :hints (("goal" :in-theory (enable inv)))
  :rule-classes :forward-chaining)

(defthm consistent-MA-p-forward
    (implies (inv MT MA)
             (consistent-MA-p MA))
  :hints (("goal" :in-theory (enable inv)))
  :rule-classes :forward-chaining)

(defthm consistent-ROB-p-forward
    (implies (consistent-MA-p MA)
             (consistent-ROB-p (MA-rob MA)))
  :hints (("goal" :in-theory (enable consistent-MA-p)))
```

```
  :rule-classes :forward-chaining)

(defthm consistent-ROB-flg-p-forward
    (implies (consistent-ROB-p ROB)
             (consistent-ROB-flg-p ROB))
  :hints (("goal" :in-theory (enable consistent-ROB-p)))
  :rule-classes :forward-chaining)

(in-theory (disable consistent-ROB-p-forward
                    consistent-ROB-flg-p-forward))

(defthm misc-inv-forward
    (implies (inv MT MA)
             (misc-inv MT MA))
  :hints (("goal" :in-theory (enable inv)))
  :rule-classes :forward-chaining)

(deflabel end-invariants-forward-lemmas)

(deftheory invariants-forward-lemmas
    (set-difference-theories (universal-theory 'end-invariants-forward-lemmas)
                     (universal-theory 'begin-invariants-forward-lemmas)))
```

D.5 Shared Lemmas

There are a couple of books to prove basic properties about the FM9801 and its MAETT. File MA2-lemmas.tex contains the proof of the properties that can be represented without the MAETT abstraction. File MAETT-lemmas1.tex and MAETT-lemmas2.tex contain a large number of basic lemmas about the FM9801. Originally, it was one book, but we divided it into two for efficiency.

D.5.1 MA2-lemmas.tex

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MA2-lemmas.lisp:
; Author  Jun Sawada, University of Texas at Austin
;
;  This file proves basic lemmas about the microarchitectural design.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(in-package "ACL2")

(include-book "utils")
(include-book "MA2-def")

(deflabel begin-MA2-lemmas)

(defthm MA-pc-MA-step
    (equal (MA-pc (MA-step MA sigs)) (step-pc MA sigs))
  :hints (("Goal" :in-theory (enable MA-step))))
```

```
(defthm MA-RF-MA-step
    (equal (MA-RF (MA-step MA sigs)) (step-RF MA))
  :hints (("Goal" :in-theory (enable MA-step))))

(defthm MA-SRF-MA-step
    (equal (MA-SRF (MA-step MA sigs)) (step-SRF MA sigs))
  :hints (("Goal" :in-theory (enable MA-step))))

(defthm MA-IFU-MA-step
    (equal (MA-IFU (MA-step MA sigs)) (step-IFU MA sigs))
  :hints (("Goal" :in-theory (enable MA-step))))

(defthm MA-DQ-MA-step
    (equal (MA-DQ (MA-step MA sigs)) (step-DQ MA sigs))
  :hints (("Goal" :in-theory (enable MA-step))))

(defthm MA-ROB-MA-step
    (equal (MA-ROB (MA-step MA sigs)) (step-ROB MA sigs))
  :hints (("Goal" :in-theory (enable MA-step))))

(defthm MA-IU-MA-step
    (equal (MA-IU (MA-step MA sigs)) (step-IU MA sigs))
  :hints (("Goal" :in-theory (enable MA-step))))

(defthm MA-MU-MA-step
    (equal (MA-MU (MA-step MA sigs)) (step-MU MA sigs))
  :hints (("Goal" :in-theory (enable MA-step))))

(defthm MA-BU-MA-step
    (equal (MA-BU (MA-step MA sigs)) (step-BU MA sigs))
  :hints (("Goal" :in-theory (enable MA-step))))

(defthm MA-LSU-MA-step
    (equal (MA-LSU (MA-step MA sigs)) (step-LSU MA sigs))
  :hints (("Goal" :in-theory (enable MA-step))))

(defthm DQ-DE0-step-DQ
    (equal (DQ-DE0 (step-DQ MA sigs))
           (step-DE0 (MA-DQ MA) MA sigs))
  :hints (("goal" :in-theory (enable step-DQ))))

(defthm DQ-DE1-step-DQ
    (equal (DQ-DE1 (step-DQ MA sigs))
           (step-DE1 (MA-DQ MA) MA sigs))
  :hints (("goal" :in-theory (enable step-DQ))))

(defthm DQ-DE2-step-DQ
    (equal (DQ-DE2 (step-DQ MA sigs))
           (step-DE2 (MA-DQ MA) MA sigs))
  :hints (("goal" :in-theory (enable step-DQ))))

(defthm DQ-DE3-step-DQ
    (equal (DQ-DE3 (step-DQ MA sigs))
           (step-DE3 (MA-DQ MA) MA sigs))
  :hints (("goal" :in-theory (enable step-DQ))))

(defthm decode-exunit-exclusive
    (and (implies (and (opcd-p op) (bitp br)
                       (b1p (logbit 0 (cntlv-exunit (decode op br)))))
                  (not (b1p (logbit 4 (cntlv-exunit (decode op br))))))
         (implies (and (opcd-p op) (bitp br)
```

```
                    (b1p (logbit 1 (cntlv-exunit (decode op br)))))
               (not (b1p (logbit 4 (cntlv-exunit (decode op br))))))
        (implies (and (opcd-p op) (bitp br)
                      (b1p (logbit 2 (cntlv-exunit (decode op br)))))
                 (not (b1p (logbit 4 (cntlv-exunit (decode op br))))))
        (implies (and (opcd-p op) (bitp br)
                      (b1p (logbit 3 (cntlv-exunit (decode op br)))))
                 (not (b1p (logbit 4 (cntlv-exunit (decode op br))))))

        (implies (and (opcd-p op) (bitp br)
                      (b1p (logbit 0 (cntlv-exunit (decode op br)))))
                 (not (b1p (logbit 3 (cntlv-exunit (decode op br))))))
        (implies (and (opcd-p op) (bitp br)
                      (b1p (logbit 1 (cntlv-exunit (decode op br)))))
                 (not (b1p (logbit 3 (cntlv-exunit (decode op br))))))
        (implies (and (opcd-p op) (bitp br)
                      (b1p (logbit 2 (cntlv-exunit (decode op br)))))
                 (not (b1p (logbit 3 (cntlv-exunit (decode op br))))))
        (implies (and (opcd-p op) (bitp br)
                      (b1p (logbit 4 (cntlv-exunit (decode op br)))))
                 (not (b1p (logbit 3 (cntlv-exunit (decode op br))))))

        (implies (and (opcd-p op) (bitp br)
                      (b1p (logbit 0 (cntlv-exunit (decode op br)))))
                 (not (b1p (logbit 2 (cntlv-exunit (decode op br))))))
        (implies (and (opcd-p op) (bitp br)
                      (b1p (logbit 1 (cntlv-exunit (decode op br)))))
                 (not (b1p (logbit 2 (cntlv-exunit (decode op br))))))
        (implies (and (opcd-p op) (bitp br)
                      (b1p (logbit 3 (cntlv-exunit (decode op br)))))
                 (not (b1p (logbit 2 (cntlv-exunit (decode op br))))))
        (implies (and (opcd-p op) (bitp br)
                      (b1p (logbit 4 (cntlv-exunit (decode op br)))))
                 (not (b1p (logbit 2 (cntlv-exunit (decode op br))))))

        (implies (and (opcd-p op) (bitp br)
                      (b1p (logbit 0 (cntlv-exunit (decode op br)))))
                 (not (b1p (logbit 1 (cntlv-exunit (decode op br))))))
        (implies (and (opcd-p op) (bitp br)
                      (b1p (logbit 2 (cntlv-exunit (decode op br)))))
                 (not (b1p (logbit 1 (cntlv-exunit (decode op br))))))
        (implies (and (opcd-p op) (bitp br)
                      (b1p (logbit 3 (cntlv-exunit (decode op br)))))
                 (not (b1p (logbit 1 (cntlv-exunit (decode op br))))))
        (implies (and (opcd-p op) (bitp br)
                      (b1p (logbit 4 (cntlv-exunit (decode op br)))))
                 (not (b1p (logbit 1 (cntlv-exunit (decode op br))))))

        (implies (and (opcd-p op) (bitp br)
                      (b1p (logbit 1 (cntlv-exunit (decode op br)))))
                 (not (b1p (logbit 0 (cntlv-exunit (decode op br))))))
        (implies (and (opcd-p op) (bitp br)
                      (b1p (logbit 2 (cntlv-exunit (decode op br)))))
                 (not (b1p (logbit 0 (cntlv-exunit (decode op br))))))
        (implies (and (opcd-p op) (bitp br)
                      (b1p (logbit 3 (cntlv-exunit (decode op br)))))
                 (not (b1p (logbit 0 (cntlv-exunit (decode op br))))))
        (implies (and (opcd-p op) (bitp br)
                      (b1p (logbit 4 (cntlv-exunit (decode op br)))))
                 (not (b1p (logbit 0 (cntlv-exunit (decode op br)))))))

:hints (("goal" :in-theory (enable decode lift-b-ops
```

```
                                    logbit* rdb))))

(defthm decode-operand-exclusive
    (and (implies (and (opcd-p op) (bitp br)
                       (b1p (logbit 0 (cntlv-operand (decode op br)))))
                  (not (b1p (logbit 3 (cntlv-operand (decode op br))))))
         (implies (and (opcd-p op) (bitp br)
                       (b1p (logbit 1 (cntlv-operand (decode op br)))))
                  (not (b1p (logbit 3 (cntlv-operand (decode op br))))))
         (implies (and (opcd-p op) (bitp br)
                       (b1p (logbit 2 (cntlv-operand (decode op br)))))
                  (not (b1p (logbit 3 (cntlv-operand (decode op br))))))

         (implies (and (opcd-p op) (bitp br)
                       (b1p (logbit 0 (cntlv-operand (decode op br)))))
                  (not (b1p (logbit 2 (cntlv-operand (decode op br))))))
         (implies (and (opcd-p op) (bitp br)
                       (b1p (logbit 1 (cntlv-operand (decode op br)))))
                  (not (b1p (logbit 2 (cntlv-operand (decode op br))))))
         (implies (and (opcd-p op) (bitp br)
                       (b1p (logbit 3 (cntlv-operand (decode op br)))))
                  (not (b1p (logbit 2 (cntlv-operand (decode op br))))))

         (implies (and (opcd-p op) (bitp br)
                       (b1p (logbit 0 (cntlv-operand (decode op br)))))
                  (not (b1p (logbit 1 (cntlv-operand (decode op br))))))
         (implies (and (opcd-p op) (bitp br)
                       (b1p (logbit 2 (cntlv-operand (decode op br)))))
                  (not (b1p (logbit 1 (cntlv-operand (decode op br))))))
         (implies (and (opcd-p op) (bitp br)
                       (b1p (logbit 3 (cntlv-operand (decode op br)))))
                  (not (b1p (logbit 1 (cntlv-operand (decode op br))))))
         (implies (and (opcd-p op) (bitp br)
                       (b1p (logbit 1 (cntlv-operand (decode op br)))))
                  (not (b1p (logbit 0 (cntlv-operand (decode op br))))))
         (implies (and (opcd-p op) (bitp br)
                       (b1p (logbit 2 (cntlv-operand (decode op br)))))
                  (not (b1p (logbit 0 (cntlv-operand (decode op br))))))
         (implies (and (opcd-p op) (bitp br)
                       (b1p (logbit 3 (cntlv-operand (decode op br)))))
                  (not (b1p (logbit 0 (cntlv-operand (decode op br)))))))

  :hints (("goal" :in-theory (enable decode lift-b-ops
                                     logbit* rdb))))

(defthm decode-cntlv-exclusive
    (and (implies (and (opcd-p op) (bitp br)
                       (b1p (cntlv-sync? (decode op br))))
                  (not (b1p (cntlv-wb? (decode op br)))))
         (implies (and (opcd-p op) (bitp br)
                       (b1p (cntlv-sync? (decode op br))))
                  (not (b1p (logbit 3 (cntlv-exunit (decode op br))))))))
  :hints (("goal" :in-theory (enable decode lift-b-ops
                                     logbit* rdb))))

(defthm dispatch-inst-if-not-DE0-valid
    (implies (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))
             (equal (dispatch-inst? MA) 0))
  :hints (("goal" :in-theory (enable dispatch-inst? lift-b-ops
                                     equal-b1p-converter dispatch-no-unit?
                                     dispatch-to-mu? dispatch-to-bu?
                                     dq-ready-no-unit? dispatch-to-iu?
```

```
                                    dq-ready-to-iu? dq-ready-to-mu?
                                    dq-ready-to-bu? dq-ready-to-lsu?
                                    dispatch-to-lsu?))))

(defthm issue-LSU-RS0-issue-LSU-RS1-exclusive
    (implies (and (MA-state-p MA) (MA-input-p sigs)
                  (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs)))
             (equal (issue-LSU-RS1? (MA-LSU MA) MA sigs) 0))
  :hints (("goal" :in-theory (enable issue-LSU-RS0? issue-LSU-RS1?
                                    lift-b-ops equal-b1p-converter
                                    LSU-RS0-ISSUE-READY?
                                    LSU-RS1-ISSUE-READY?))))

(defthm cases-of-CDB-ready
    (implies (b1p (CDB-ready? MA))
             (or (b1p (CDB-for-IU? MA))
                 (b1p (CDB-for-BU? MA))
                 (b1p (CDB-for-MU? MA))
                 (b1p (CDB-for-LSU? MA))))
  :hints (("goal" :in-theory (enable CDB-ready? lift-b-ops
                                    CDB-for-LSU? CDB-for-IU?
                                    CDB-for-BU? CDB-for-MU?)))
  :rule-classes nil)

(defthm cases-of-CDB-ready-for
    (implies (b1p (CDB-ready-for? rix MA))
             (or (b1p (CDB-for-IU? MA))
                 (b1p (CDB-for-BU? MA))
                 (b1p (CDB-for-MU? MA))
                 (b1p (CDB-for-LSU? MA))))
  :hints (("goal" :in-theory (enable CDB-ready-for? CDB-ready?
                                    lift-b-ops
                                    CDB-for-LSU? CDB-for-IU?
                                    CDB-for-BU? CDB-for-MU?)))
  :rule-classes nil)

(defthm CDB-for-exclusive
    (and (implies (b1p (CDB-for-IU? MA)) (not (b1p (CDB-for-BU? MA))))
         (implies (b1p (CDB-for-IU? MA)) (not (b1p (CDB-for-MU? MA))))
         (implies (b1p (CDB-for-IU? MA)) (not (b1p (CDB-for-LSU? MA))))
         (implies (b1p (CDB-for-BU? MA)) (not (b1p (CDB-for-IU? MA))))
         (implies (b1p (CDB-for-BU? MA)) (not (b1p (CDB-for-MU? MA))))
         (implies (b1p (CDB-for-BU? MA)) (not (b1p (CDB-for-LSU? MA))))
         (implies (b1p (CDB-for-MU? MA)) (not (b1p (CDB-for-IU? MA))))
         (implies (b1p (CDB-for-MU? MA)) (not (b1p (CDB-for-BU? MA))))
         (implies (b1p (CDB-for-MU? MA)) (not (b1p (CDB-for-LSU? MA))))
         (implies (b1p (CDB-for-LSU? MA)) (not (b1p (CDB-for-IU? MA))))
         (implies (b1p (CDB-for-LSU? MA)) (not (b1p (CDB-for-BU? MA))))
         (implies (b1p (CDB-for-LSU? MA)) (not (b1p (CDB-for-MU? MA)))))
  :hints (("goal" :in-theory (enable lift-b-ops cdb-def))))

(defthm len-MA-ROB-entries
    (implies (MA-state-p MA)
             (equal (len (ROB-entries (MA-ROB MA))) *ROB-size*))
  :hints (("goal" :in-theory (enable MA-state-p ROB-p ROB-entries-p))))

(encapsulate nil
(local
 (defun induct-nth-robe (entries idx1 idx2)
     (if (endp entries)  nil
         (if (zp idx1) (list entries idx1 idx2)
             (induct-nth-robe (cdr entries) (1- idx1) (1+ idx2)))))))
```

```
(local
 (defthm nth-robe-step-robe-help
     (implies (and (integerp idx1) (<= 0 idx1)
                   (integerp idx2) (<= 0 idx2)
                   (< (+ idx1 idx2) 8)
                   (< idx1 (len entries)))
              (equal (nth idx1 (step-robe-list entries idx2 ROB MA sigs))
                     (step-robe (nth idx1 entries) (+ idx1 idx2) ROB MA sigs)))
   :hints (("goal" :induct (induct-nth-robe entries idx1 idx2)
                   :in-theory (enable STEP-ROBE-LIST unsigned-byte-p
                                      rob-index-p))
           (when-found (STEP-ROBE-LIST ENTRIES IDX2 ROB MA SIGS)
                       (:expand (STEP-ROBE-LIST ENTRIES IDX2 ROB MA SIGS))))))

; A lemma to help opening the definition of step-ROB.
(defthm nth-robe-step-rob
    (implies (and (MA-state-p MA) (rob-index-p index))
             (equal (nth-robe index (step-ROB MA sigs))
                    (step-robe (nth-robe index (MA-rob MA))
                               index (MA-rob MA) MA sigs)))
  :hints (("goal" :in-theory (enable step-rob nth-robe step-rob-entries))))
)

(defthm not-ex-intr-if-not-exintr-flag-nor-input-exintr
    (implies (and (MA-state-p MA) (MA-input-p sigs)
                  (not (b1p (MA-input-exintr sigs)))
                  (not (b1p (exintr-flag? MA))))
             (equal (ex-intr? MA sigs) 0))
  :hints (("Goal" :in-theory (enable exintr-flag? ex-intr? lift-b-ops
                                     b1p-bit-rewriter))))

(defthm not-ex-intr-if-rob-not-empty
    (implies (not (b1p (rob-empty? (MA-rob MA))))
             (equal (ex-intr? MA sigs) 0))
  :hints (("Goal" :in-theory (enable ex-intr? lift-b-ops
                                     b1p-bit-rewriter))))

(defthm commit-inst-if-leave-excpt
    (implies (and (MA-state-p MA) (b1p (leave-excpt? MA)))
             (equal (commit-inst? MA) 1))
  :hints (("goal" :in-theory (enable lift-b-ops leave-excpt? commit-inst?
                                     equal-b1p-converter
                                     b1p-bit-rewriter))))

(defthm commit-inst-if-enter-excpt
    (implies (and (MA-state-p MA) (b1p (enter-excpt? MA)))
             (equal (commit-inst? MA) 1))
  :hints (("goal" :in-theory (enable lift-b-ops enter-excpt? commit-inst?
                                     equal-b1p-converter
                                     b1p-bit-rewriter))))

(defthm cntlv-exunit-decode
    (implies (and (syntaxp (quoted-constant-p op)) (bitp br))
             (equal (cntlv-exunit (decode op br))
                    (cntlv-exunit (decode op 0))))
  :hints (("goal" :in-theory (enable decode logbit* rdb lift-b-ops
                                     loghead* logtail*))))

(defthm cntlv-operand-decode
    (implies (and (syntaxp (quoted-constant-p op)) (bitp br))
```

```
            (equal (cntlv-operand (decode op br))
                   (cntlv-operand (decode op 0))))
  :hints (("goal" :in-theory (enable decode logbit* rdb lift-b-ops
                                     loghead* logtail*))))

(defthm cntlv-br-predict-decode
    (implies (bitp br)
             (equal (cntlv-br-predict? (decode op br))
                    br))
  :hints (("goal" :in-theory (enable decode logbit* rdb lift-b-ops
                                     loghead* logtail*))))

(defthm cntlv-ld-st-decode
    (implies (syntaxp (quoted-constant-p op))
             (equal (cntlv-ld-st? (decode op br))
                    (cntlv-ld-st? (decode op 0))))
  :hints (("goal" :in-theory (enable decode logbit* rdb lift-b-ops))))

(defthm cntlv-wb-decode
    (implies (syntaxp (quoted-constant-p op))
             (equal (cntlv-wb? (decode op br))
                    (cntlv-wb? (decode op 0))))
  :hints (("goal" :in-theory (enable decode logbit* rdb lift-b-ops))))

(defthm cntlv-wb-sreg-decode
    (implies (syntaxp (quoted-constant-p op))
             (equal (cntlv-wb-sreg? (decode op br))
                    (cntlv-wb-sreg? (decode op 0))))
  :hints (("goal" :in-theory (enable decode logbit* rdb lift-b-ops))))

(defthm cntlv-sync-decode
    (implies (syntaxp (quoted-constant-p op))
             (equal (cntlv-sync? (decode op br))
                    (cntlv-sync? (decode op 0))))
  :hints (("goal" :in-theory (enable decode logbit* rdb lift-b-ops))))

(defthm cntlv-rfeh-decode
    (implies (syntaxp (quoted-constant-p op))
             (equal (cntlv-rfeh? (decode op br))
                    (cntlv-rfeh? (decode op 0))))
  :hints (("goal" :in-theory (enable decode logbit* rdb lift-b-ops))))

(defthm cntlv-sync-and-branch-of-decode-exclusive
    (implies (and (opcd-p op) (bitp flg)
                  (b1p (cntlv-sync? (decode op flg))))
             (equal (logbit 3 (cntlv-exunit (decode op flg))) 0))
  :hints (("goal" :in-theory (enable decode lift-b-ops
                                     equal-b1p-converter
                                     rdb logbit*
                                     opcd-p
                                     bv-eqv-iff-equal))))

(defthm exintr-flag-ma-step
    (implies (and (MA-state-p MA) (MA-input-p sigs)
                  (not (b1p (exintr-flag? MA)))
                  (not (b1p (MA-input-exintr sigs))))
             (not (b1p (exintr-flag? (MA-step MA sigs)))))
  :hints (("goal" :in-theory (enable exintr-flag? MA-step step-rob
                                     lift-b-ops))))

; When the ROB is full, no instruction is dispatched.
(defthm not-dispatch-inst-if-rob-full
```

```
    (implies (and (MA-state-p MA) (b1p (ROB-full? (MA-rob MA))))
             (equal (dispatch-inst? MA) 0))
  :hints (("goal" :in-theory (enable lift-b-ops MA-def equal-b1p-converter))))
```

D.5.2 MAETT-lemmas1.tex

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MAETT-lemmas1.lisp
; Author  Jun Sawada, University of Texas at Austin
;
;  This book contains various lemmas about the FM9801 and its MAETT
;  abstraction.  This book continues to another book MAETT-lemmas2.lisp.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(in-package "ACL2")

(include-book "MA2-lemmas")
(include-book "invariants-def")
(deflabel begin-MAETT-lemmas1)

; Index
;   INST-in subtrace-p INST-in-order and related lemmas
;   Def and lemmas about ISA-before specultv-before?, modified-inst-before?
;   and ISA-before.
;   Redefining MA-stepn
;   Misc Lemmas
;   Lemmas about stages
;     Basic Lemmas
;     Theory  about  dispatched and committed instructions
;   Lemmas about init-MT
;   Lemmas about micro-architecture
;   Lemmas about stages after step-INST
;   Lemmas about INST-functions and step-INST
;   Lemmas about relations between instructions
;   Lemmas to open INST-inv
;     Lemmas after this are in MAETT-lemmas2.lisp
;   Lemmas about relations between MT and MA
;     relations between exception events
;     (this part occupies 60% of the whole file)
;   Lemmas about micro-architecture satisfying invariants.
;   Lemmas about stages after step-INST, again.
;   Lemmas about commit again.
;   Lemmas about no-commit-inst-p and no-dispatched-inst-p
;   Lemmas about MT-no-jmp-exintr-before

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   INST-in subtrace-p INST-in-order and related lemmas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun head-p (x y)
  (if (endp x)
      t
      (if (endp y)
          nil
          (and (equal (car x) (car y))
               (head-p (cdr x) (cdr y))))))

(defun tail-p (sublist list)
  (declare (xargs :guard (and (true-listp sublist) (true-listp list))))
  (if (equal sublist list)
      T
```

```
      (if (endp list)
          nil
          (tail-p sublist (cdr list)))))

(defun tail-after-p (elm sub lst)
  (declare (xargs :guard (and (true-listp sub) (true-listp lst))))
  (tail-p sub (cdr (member-equal elm lst))))

(defun member-in-order (elm1 elm2 lst)
  (declare (xargs :guard (true-listp lst)))
  (member-equal elm2 (cdr (member-equal elm1 lst))))

(defthm member-equal-car-of-tail-p
    (implies (and (consp sub) (tail-p sub lst))
             (member-equal (car sub) lst)))

(defthm tail-p-cdr-1
    (implies (and (consp sub) (tail-p sub lst))
             (tail-p (cdr sub) lst)))

(defthm tail-p-nil
    (implies (true-listp list)
             (tail-p nil list)))

(defthm tail-p-transitivity
    (implies (and (tail-p lst1 lst2) (tail-p lst2 lst3))
             (tail-p lst1 lst3)))

(defthm tail-p-member-equal
    (implies (and (tail-p lst1 lst2) (member-equal elm lst1))
             (member-equal elm lst2)))

(in-theory (disable tail-p-transitivity tail-p-member-equal))

(defthm tail-p-cdr-2
    (implies (consp lst) (not (tail-p lst (cdr lst))))
  :hints (("Goal" :induct (len lst))))

(defthm tail-antisymmetry
    (implies (and (not (equal x y)) (tail-p x y))
             (not (tail-p y x)))
  :rule-classes nil)

(defthm tail-p-antisymmetry
    (implies (and (tail-p sub lst) (not (equal sub lst)))
             (not (tail-p lst sub))))

(defthm tail-p-cddr
    (implies (consp (cdr lst))
             (not (tail-p lst (cddr lst)))))

(defthm tail-after-p*
    (equal (tail-after-p elm sub lst)
           (if (endp lst)
               (null sub)
               (if (equal elm (car lst))
                   (tail-p sub (cdr lst))
                   (tail-after-p elm sub (cdr lst))))))
  :rule-classes :definition)
(in-theory (disable tail-after-p*))

(defthm tail-after-p-nil
```

```
    (implies (true-listp lst)
             (tail-after-p elm nil lst)))

(defthm tail-after-p-cdr
    (implies (and (consp sub) (tail-after-p elm sub lst))
             (tail-after-p elm (cdr sub) lst)))

(defthm tail-after-p-car-cdr
    (implies (and (consp sub) (tail-p sub lst))
             (tail-after-p (car sub) (cdr sub) lst))
  :hints (("goal" :in-theory (enable tail-after-p*))))

(in-theory (disable tail-after-p member-in-order))

(defun INST-in (i MT)
  (declare (xargs :guard (and (INST-p i) (MAETT-p MT))))
  (member-equal i (MT-trace MT)))

(defun subtrace-p (trace MT)
  (declare (xargs :guard (and (INST-listp trace) (MAETT-p MT))))
  (tail-p trace (MT-trace MT)))

(defun subtrace-after-p (elm sub MT)
  (declare (xargs :guard (and (INST-p elm) (INST-listp sub) (MAETT-p MT))))
  (tail-after-p elm sub (MT-trace MT)))

(defun INST-in-order-p (i1 i2 MT)
  (declare (xargs :guard (and (INST-p i1) (INST-p i2)
                              (MAETT-p MT))))
  (member-in-order i1 i2 (MT-trace MT)))

(defthm INST-in-car-of-subtrace
    (implies (and (consp trace) (subtrace-p trace MT))
             (INST-in (car trace) MT)))

(defthm member-of-subtrace
    (implies (and (member-equal i trace)
                  (subtrace-p trace MT))
             (INST-in i MT)))

(defthm member-in-order*
    (equal (member-in-order elm1 elm2 lst)
           (if (endp lst)
               nil
               (if (equal (car lst) elm1)
                   (member-equal elm2 (cdr lst))
                   (member-in-order elm1 elm2 (cdr lst)))))
  :hints (("Goal" :in-theory (enable member-in-order)))
  :rule-classes :definition)

(in-theory (disable member-in-order*))

(defthm not-member-in-order-if-member-equal-1
    (implies (not (member-equal elm1 trace))
             (not (member-in-order elm1 elm2 trace)))
  :hints (("Goal" :in-theory (enable member-in-order*))))

(defthm not-member-in-order-if-member-equal-2
    (implies (not (member-equal elm2 trace))
             (not (member-in-order elm1 elm2 trace)))
  :hints (("Goal" :in-theory (enable member-in-order*))))
```

```
(defthm subtrace-p-MT-trace
    (subtrace-p (MT-trace MT) MT))

(defthm subtrace-p-nil
    (implies (MAETT-p MT) (subtrace-p nil MT)))

(defthm subtrace-p-cdr
    (implies (and (consp trace) (subtrace-p trace MT))
             (subtrace-p (cdr trace) MT)))

(defthm subtrace-tail-p
    (implies (and (tail-p sub trace) (subtrace-p trace MT))
             (subtrace-p sub MT))
  :hints (("goal" :in-theory (enable subtrace-p))))

(defthm subtrace-after-p-nil
    (implies (MAETT-p MT)
             (subtrace-after-p i nil MT)))

(defthm subtrace-after-p-cdr
    (implies (and (consp trace) (subtrace-after-p i trace MT))
             (subtrace-after-p i (cdr trace) MT)))

(defthm subtrace-after-p-car-cdr
    (implies (and (consp trace) (subtrace-p trace MT))
             (subtrace-after-p (car trace) (cdr trace) MT)))

(encapsulate nil
(local
(defthm subtrace-p-subtrace-after-p-help
    (implies (and (INST-listp trace) (tail-after-p i sub trace))
             (tail-p sub trace))
  :hints (("goal" :in-theory (enable tail-after-p*)))))

(defthm subtrace-p-subtrace-after-p
    (implies (and (MAETT-p MT) (subtrace-after-p i trace MT))
             (subtrace-p trace MT))
  :hints (("goal" :in-theory (enable subtrace-after-p subtrace-p))))
)

(encapsulate nil
(local
(defthm INST-in-order-car-if-subtrace-after-p-help
    (implies (and (consp sub) (tail-after-p i sub trace))
             (member-in-order i (car sub) trace))
  :hints (("goal" :in-theory (enable tail-after-p* member-in-order*)
                  :induct (member-equal i trace)))))

(defthm INST-in-order-car-if-subtrace-after-p
    (implies (and (consp sub) (subtrace-after-p i sub MT))
             (INST-in-order-p i (car sub) MT))
  :hints (("goal" :in-theory (enable subtrace-after-p INST-in-order-p))))
)

(encapsulate nil
(local
(defthm not-member-equal-car-help
    (implies (and (distinct-member-p trace)
                  (tail-p sub trace))
             (not (member-equal (car sub) (cdr sub))))))

(defthm not-member-equal-car
```

```
    (implies (and (inv MT MA)
                  (subtrace-p trace MT))
             (not (member-equal (car trace) (cdr trace))))
  :hints (("goal" :in-theory (enable inv weak-inv
                                     MT-distinct-inst-p subtrace-p)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (subtrace-p trace MT)
                           (member-equal i (cdr trace)))
                      (not (equal (car trace) i))))
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (subtrace-p trace MT)
                           (member-equal i (cdr trace)))
                      (not (equal i (car trace))))))
)
)

(encapsulate nil
(local
(defthm INST-in-order-p-car-trace-help
    (implies (and (consp trace)
                  (tail-p trace trace2) (member-equal i trace)
                  (not (equal i (car trace))))
             (member-in-order (car trace) i trace2))
  :hints (("goal" :in-theory (enable member-in-order*)))))

(defthm INST-in-order-p-car-trace
    (implies (and (consp trace)
                  (subtrace-p trace MT) (member-equal i trace)
                  (not (equal i (car trace))))
             (INST-in-order-p (car trace) i MT))
  :hints (("Goal" :in-theory (enable INST-in-order-p subtrace-p)
                  :do-not-induct t)))
)

(encapsulate nil
(local
(defthm member-in-order-antisymmetry
    (implies (and (distinct-member-p trace)
                  (member-in-order i j trace))
             (not (member-in-order j i trace)))
  :hints (("goal" :in-theory (enable member-in-order*)))))

(defthm INST-in-order-antisymmetry
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in-order-p i j MT))
             (not (INST-in-order-p j i MT)))
  :Hints (("Goal" :in-theory (enable INST-in-order-p inv
                                     weak-inv
                                     MT-distinct-inst-p))))
)

(encapsulate nil
(local
(defthm INST-in-order-p-total-help
    (implies (and (not (member-in-order j i trace))
                  (not (equal i j))
                  (member-equal i trace) (member-equal j trace))
```

```
            (member-in-order i j trace))
  :hints (("goal" :in-theory (enable member-in-order*)))))

(defthm INST-in-order-p-total
    (implies (and (not (INST-in-order-p j i MT))
                  (not (equal i j))
                  (INST-in i MT) (INST-in j MT))
             (INST-in-order-p i j MT))
  :hints (("goal" :in-theory (enable INST-in INST-in-order-p))))
)

(encapsulate nil
(local
 (defthm member-in-order-transitivity
     (implies (and (distinct-member-p trace)
                   (member-in-order i j trace)
                   (member-in-order j k trace))
              (member-in-order i k trace))
   :hints (("Goal" :in-theory (enable member-in-order*)
                   :induct (len trace)))))

(defthm INST-in-order-transitivity
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in-order-p i j MT)
                  (INST-in-order-p j k MT))
             (INST-in-order-p i k MT))
  :hints (("goal" :in-theory (enable inv weak-inv
                                     MT-distinct-inst-p
                                     INST-in-order-p)))
  :rule-classes
  ((:rewrite :corollary
             (implies (and (INST-in-order-p j k MT)
                           (inv MT MA)
                           (MAETT-p MT) (MA-state-p MA)
                           (INST-in-order-p i j MT))
                      (INST-in-order-p i k MT)))
   (:rewrite :corollary
             (implies (and (INST-in-order-p i j MT)
                           (inv MT MA)
                           (MAETT-p MT) (MA-state-p MA)
                           (INST-in-order-p j k MT))
                      (INST-in-order-p i k MT)))))
)

(in-theory (disable INST-in subtrace-p subtrace-after-p INST-in-order-p))

(defthm member-in-order-car-member-cdr
    (implies (and (tail-p sub trace)
                  (member-equal i (cdr sub)))
             (member-in-order (car sub) i trace))
  :hints (("goal" :in-theory (enable member-in-order*))))

; If instruction i is a member or (cdr trace), i follows instruction
; (car trace).
(defthm INST-in-order-car-member-cdr
    (implies (and (subtrace-p trace MT)
                  (member-equal i (cdr trace)))
             (INST-in-order-p (car trace) i MT))
  :hints (("goal" :in-theory (enable subtrace-p INST-in-order-p))))

(encapsulate nil
```

```
(local
(defthm INST-in-order-p-cars-if-tailp-help
    (implies (and (tail-p lst2 trace)
                  (tail-p lst1 lst2)
                  (consp lst1)
                  (not (equal lst1 lst2)))
             (member-in-order (car lst2) (car lst1) trace))
  :hints (("Goal" :in-theory (enable member-in-order*)))))

(defthm INST-in-order-p-cars-if-tailp
    (implies (and (subtrace-p trace MT)
                  (tail-p sub trace)
                  (consp sub)
                  (not (equal sub trace)))
             (INST-in-order-p (car trace) (car sub) MT)))
)

(encapsulate nil
(local
(defthm INST-in-order-p-identity-help
    (implies (and (DISTINCT-MEMBER-P trace)
                  (member-equal i trace))
             (not (member-in-order i i trace)))
  :hints (("Goal" :in-theory (enable member-in-order*)))))

(defthm INST-in-order-p-identity
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA))
             (not (INST-in-order-p i i MT)))
  :hints (("Goal" :in-theory (enable INST-in INST-in-order-p
                                     inv weak-inv
                                     MT-distinct-inst-p))))
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Speculative, modified-inst, pre-ISA before a terminal subtrace.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun specultv-at-tail (sub trace spc)
  (declare (xargs :guard (and (INST-listp sub) (INST-listp trace)
                              (bitp spc))))
  (if (equal sub trace)
      spc
      (if (endp trace) spc
          (specultv-at-tail sub (cdr trace)
                            (b-ior (inst-specultv? (car trace))
                                   (INST-start-specultv? (car trace)))))))

(defthm bitp-specultv-at-tail
    (implies (bitp spc)
             (bitp (specultv-at-tail sub trace spc))))

(defun specultv-before? (sub MT)
  (declare (xargs :guard (and (INST-listp sub) (MAETT-p MT))))
  (specultv-at-tail sub (MT-trace MT) 0))

(defthm bitp-specultv-before?
    (bitp (specultv-before? sub MT)))

(defun modified-inst-before-tail (sub trace smc)
  (declare (xargs :guard (and (INST-listp sub) (INST-listp trace)
                              (bitp smc))))
  (if (equal sub trace)
```

```
      smc
      (if (endp trace) smc
          (modified-inst-before-tail sub (cdr trace)
                                     (INST-modified? (car trace))))))

(defthm bitp-modified-inst-before-tail
    (implies (and (INST-listp trace) (bitp smc))
             (bitp (modified-inst-before-tail sub trace smc))))

(defun modified-inst-before? (sub MT)
  (declare (xargs :guard (and (INST-listp sub) (MAETT-p MT))))
  (modified-inst-before-tail sub (MT-trace MT) 0))

(defthm bitp-modified-inst-before?
    (implies (MAETT-p MT) (bitp (modified-inst-before? sub MT))))

(defun ISA-at-tail (sub trace pre)
  (declare (xargs :guard (and (INST-listp sub) (INST-listp trace)
                              (ISA-state-p pre))))
  (if (equal sub trace)
      pre
      (if (endp trace) pre
          (ISA-at-tail sub (cdr trace) (INST-post-ISA (car trace))))))

(defthm ISA-state-p-ISA-at-tail
    (implies (and (ISA-state-p pre)
                  (INST-listp trace))
             (ISA-state-p (ISA-at-tail sub trace pre))))

(defun ISA-before (sub MT)
  (declare (xargs :guard (and (INST-listp sub) (MAETT-p MT))))
  (ISA-at-tail sub (MT-trace MT) (MT-init-ISA MT)))

(defthm ISA-state-p-ISA-before
    (implies (MAETT-p MT)
             (ISA-state-p (ISA-before sub MT))))

(in-theory (disable ISA-before specultv-before? modified-inst-before?))

(encapsulate nil
(local
(defthm specultv-before-cdr-help
    (implies (and (tail-p sub trace)
                  (consp sub))
             (equal (specultv-at-tail (cdr sub) trace spc)
                    (b-ior (inst-specultv? (car sub))
                           (INST-start-specultv? (car sub)))))))

(defthm specultv-before-cdr
    (implies (and (subtrace-p sub MT)
                  (consp sub))
             (equal (specultv-before? (cdr sub) MT)
                    (b-ior (inst-specultv? (car sub))
                           (INST-start-specultv? (car sub)))))
  :hints (("Goal" :in-theory (enable specultv-before?
                                     subtrace-p
                                     ISA-STEP-CHAIN-P))))
)

(encapsulate nil
(local
(defthm specultv-at-tail-nil-if-trace-all-specultv
```

```
    (implies (and (trace-all-specultv-p trace)
                  (b1p spc))
             (b1p (specultv-at-tail sub trace spc)))
  :hints (("Goal" :in-theory (enable lift-b-ops)))))

(local
(defthm specultv-at-tail-nil
    (implies (and (trace-correct-speculation-p trace)
                  (bitp spc) (not (b1p spc)))
             (equal (specultv-at-tail nil trace spc)
                    (trace-specultv? trace)))
  :hints (("Goal" :in-theory (enable lift-b-ops equal-b1p-converter)))))

(defthm specultv-before-nil
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA))
             (equal (specultv-before? nil MT)
                    (MT-specultv? MT)))
  :hints (("Goal" :in-theory (enable specultv-before? MT-specultv?
                                     inv
                                     weak-inv correct-speculation-p))))
)

(defthm specultv-before-MT-trace
    (equal (specultv-before? (MT-trace MT) MT) 0)
  :hints (("Goal" :in-theory (enable specultv-before?))))

(encapsulate nil
(local
(defthm modified-inst-before-cdr-help
    (implies (and (tail-p sub trace)
                  (consp sub))
             (equal (modified-inst-before-tail (cdr sub) trace spc)
                    (INST-modified? (car sub))))))

(defthm modified-inst-before-cdr
    (implies (and (subtrace-p sub MT)
                  (consp sub))
             (equal (modified-inst-before? (cdr sub) MT)
                    (INST-modified? (car sub))))
  :hints (("Goal" :in-theory (enable modified-inst-before? weak-inv
                                     inv subtrace-p
                                     ISA-STEP-CHAIN-P))))
)

(encapsulate nil
(local
(defthm modified-inst-before-tail-nil-if-trace-is-all-modified-flgs-on
    (implies (and (trace-correct-modified-flgs-p trace MT 1)
                  (b1p spc))
             (b1p (modified-inst-before-tail sub trace spc)))
  :hints (("Goal" :in-theory (enable lift-b-ops)))))

(local
(defthm modified-inst-before-nil-help
    (implies (and (trace-correct-modified-flgs-p trace MT 0)
                  (INST-listp trace)
                  (bitp smc)
                  (not (b1p smc)))
             (equal (modified-inst-before-tail nil trace smc)
                    (trace-self-modify? trace)))
```

```
  :hints (("Goal" :in-theory (enable lift-b-ops equal-b1p-converter)))))

(defthm modified-inst-before-nil
    (implies (and (weak-inv MT)
                  (MAETT-p MT))
             (equal (modified-inst-before? nil MT)
                    (MT-self-modify? MT)))
  :hints (("Goal" :in-theory (enable modified-inst-before? MT-self-modify?
                                     weak-inv correct-modified-flgs-p)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (MAETT-p MT)
                           (MA-state-p MA))
                      (equal (modified-inst-before? nil MT)
                             (MT-self-modify? MT))))))

)

(defthm modified-inst-before-MT-trace
    (equal (modified-inst-before? (MT-trace MT) MT) 0)
  :hints (("Goal" :in-theory (enable modified-inst-before?))))

(encapsulate nil
(local
(defthm ISA-before-cdr-help
    (implies (and (ISA-chained-trace-p trace pre)
                  (tail-p sub trace)
                  (consp sub))
             (equal (ISA-at-tail (cdr sub) trace pre)
                    (INST-post-ISA (car sub))))))

(defthm ISA-before-cdr
    (implies (and (weak-inv MT)
                  (subtrace-p sub MT)
                  (consp sub))
             (equal (ISA-before (cdr sub) MT)
                    (INST-post-ISA (car sub))))
  :hints (("Goal" :in-theory (enable ISA-before weak-inv inv
                                     subtrace-p
                                     ISA-STEP-CHAIN-P)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (subtrace-p sub MT)
                           (consp sub))
                      (equal (ISA-before (cdr sub) MT)
                             (INST-post-ISA (car sub)))))))
)

(defthm ISA-at-tail-nil
    (equal (ISA-at-tail nil trace pre)
           (trace-final-ISA trace pre)))

(defthm ISA-before-nil
    (equal (ISA-before nil MT)
           (MT-final-ISA MT))
  :hints (("Goal" :in-theory (enable ISA-before MT-final-ISA))))

(defthm ISA-before-MT-trace
```

```
    (equal (ISA-before (MT-trace MT) MT)
           (MT-init-ISA MT))
  :hints (("Goal" :in-theory (enable ISA-before))))

(encapsulate nil
(local
(defthm ISA-before-MT-non-nil-trace-help
    (implies (and (consp sub)
                  (tail-p sub trace)
                  (ISA-chained-trace-p trace pre))
             (equal (ISA-at-tail sub trace pre)
                    (INST-pre-ISA (car sub))))))

(defthm ISA-before-MT-non-nil-trace
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (consp trace)
                  (subtrace-p trace MT))
             (equal (ISA-before trace MT)
                    (INST-pre-ISA (car trace))))
  :hints (("Goal" :in-theory (enable weak-inv inv
                                     ISA-step-chain-p
                                     subtrace-p ISA-before))))
)
(in-theory (disable ISA-before-MT-non-nil-trace))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Redefining MA-stepn
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defthm MA-step0
    (equal (MA-stepn MA sig-lst 0) MA)
  :hints (("goal" :in-theory (enable MA-stepn))))

(defthm MA-stepn*
    (implies (<= n (len sig-lst))
             (equal (MA-stepn MA sig-lst n)
                    (if (zp n)
                        MA
                        (MA-step (MA-stepn MA sig-lst (1- n))
                                 (nth (1- n) sig-lst))))))
  :hints (("Goal" :in-theory (enable MA-stepn)))
  :rule-classes :definition)

(in-theory (disable MA-stepn*))

(defthm MT-stepn*
    (equal (MT-stepn MT MA sig-lst n)
           (if (zp n)
               MT
               (if (< (len sig-lst) n)
                   (MT-stepn MT MA sig-lst (1- n))
                   (MT-step (MT-stepn MT MA sig-lst (1- n))
                            (MA-stepn MA sig-lst (1- n))
                            (nth (1- n) sig-lst))))))
  :hints (("Goal" :in-theory (enable MA-stepn)))
  :rule-classes :definition)

(in-theory (disable MT-stepn*))

(defthm MT-trace-MT-step
    (equal (MT-trace (MT-step MT MA sigs))
```

```
          (step-trace (MT-trace MT) MT MA sigs (MT-init-ISA MT) 0 0))
  :hints (("goal" :in-theory (enable MT-step))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Miscellaneous Lemmas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defthm rob-index-p-plus
    (implies (and (<= x 0) (<= 0 (+ x y))
                  (rob-index-p y)
                  (integerp x))
             (rob-index-p (+ x y)))
  :hints (("goal" :in-theory (enable rob-index-p unsigned-byte-p))))

(defthm ISA-extensionality
    (implies (and (ISA-state-p s1)
                  (ISA-state-p s2)
                  (equal (ISA-pc s1) (ISA-pc s2))
                  (equal (ISA-RF s1) (ISA-RF s2))
                  (equal (ISA-SRF s1) (ISA-SRF s2))
                  (equal (ISA-mem s1) (ISA-mem s2)))
             (equal s1 s2))
  :rule-classes
  ((:rewrite :corollary
             (implies (and (ISA-state-p s1)
                           (ISA-state-p s2)
                           (equal (ISA-pc s1) (ISA-pc s2))
                           (equal (ISA-RF s1) (ISA-RF s2))
                           (equal (ISA-SRF s1) (ISA-SRF s2))
                           (equal (ISA-mem s1) (ISA-mem s2)))
                      (equal (equal s1 s2) t)))))

(in-theory (disable ISA-extensionality))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Theories about stages
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defthm INST-is-at-one-of-the-stages
    (implies (INST-p i)
             (or (IFU-stg-p (INST-stg i))
                 (DQ-stg-p (INST-stg i))
                 (EXECUTE-stg-p (INST-stg i))
                 (COMPLETE-stg-p (INST-stg i))
                 (COMMIT-stg-p (INST-stg i))
                 (RETIRE-stg-p (INST-stg i))))
  :hints (("goal" :in-theory (enable MA-stg-def)))
  :rule-classes
  ((:rewrite :corollary
            (implies (and (INST-p i)
                  (not (IFU-stg-p (INST-stg i)))
                  (not (DQ-stg-p (INST-stg i)))
                  (not (EXECUTE-stg-p (INST-stg i)))
                  (not (COMPLETE-stg-p (INST-stg i)))
                  (not (COMMIT-stg-p (INST-stg i))))
             (RETIRE-stg-p (INST-stg i))))))

(defthm IFU-stg-not-DQ-stg
    (implies (IFU-stg-p stg) (not (DQ-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm IFU-stg-not-execute-stg
    (implies (IFU-stg-p stg) (not (execute-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))
```

```
(defthm IFU-stg-not-complete-stg
    (implies (IFU-stg-p stg) (not (complete-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm IFU-stg-not-commit-stg
    (implies (IFU-stg-p stg) (not (commit-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm IFU-stg-not-retire-stg
    (implies (IFU-stg-p stg) (not (retire-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm DQ-stg-not-IFU-stg
    (implies (DQ-stg-p stg) (not (IFU-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm DQ-stg-not-execute-stg
    (implies (DQ-stg-p stg) (not (execute-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm DQ-stg-not-complete-stg
    (implies (DQ-stg-p stg) (not (complete-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm DQ-stg-not-commit-stg
    (implies (DQ-stg-p stg) (not (commit-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm DQ-stg-not-retire-stg
    (implies (DQ-stg-p stg) (not (retire-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm execute-stg-not-IFU-stg
    (implies (execute-stg-p stg) (not (IFU-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm execute-stg-not-DQ-stg
    (implies (execute-stg-p stg) (not (DQ-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm execute-stg-not-complete-stg
    (implies (execute-stg-p stg) (not (complete-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm execute-stg-not-commit-stg
    (implies (execute-stg-p stg) (not (commit-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm execute-stg-not-retire-stg
    (implies (execute-stg-p stg) (not (retire-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm complete-stg-not-IFU-stg
    (implies (complete-stg-p stg) (not (IFU-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm complete-stg-not-DQ-stg
    (implies (complete-stg-p stg) (not (DQ-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm complete-stg-not-execute-stg
```

```
    (implies (complete-stg-p stg) (not (execute-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm complete-stg-not-commit-stg
    (implies (complete-stg-p stg) (not (commit-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm complete-stg-not-retire-stg
    (implies (complete-stg-p stg) (not (retire-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm commit-stg-not-IFU-stg
    (implies (commit-stg-p stg) (not (IFU-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm commit-stg-not-DQ-stg
    (implies (commit-stg-p stg) (not (DQ-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm commit-stg-not-execute-stg
    (implies (commit-stg-p stg) (not (execute-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm commit-stg-not-complete-stg
    (implies (commit-stg-p stg) (not (complete-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm commit-stg-not-retire-stg
    (implies (commit-stg-p stg) (not (retire-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm retire-stg-not-IFU-stg
    (implies (retire-stg-p stg) (not (IFU-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm retire-stg-not-DQ-stg
    (implies (retire-stg-p stg) (not (DQ-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm retire-stg-not-execute-stg
    (implies (retire-stg-p stg) (not (execute-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm retire-stg-not-complete-stg
    (implies (retire-stg-p stg) (not (complete-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm retire-stg-not-commit-stg
    (implies (retire-stg-p stg) (not (commit-stg-p stg)))
  :hints (("Goal" :in-theory (enable MA-stg-def))))

(defthm execute-stage-exclusive
    (and (implies (LSU-stg-p stg) (not (IU-stg-p stg)))
         (implies (LSU-stg-p stg) (not (MU-stg-p stg)))
         (implies (LSU-stg-p stg) (not (BU-stg-p stg)))
         (implies (IU-stg-p stg) (not (LSU-stg-p stg)))
         (implies (IU-stg-p stg) (not (MU-stg-p stg)))
         (implies (IU-stg-p stg) (not (BU-stg-p stg)))
         (implies (BU-stg-p stg) (not (LSU-stg-p stg)))
         (implies (BU-stg-p stg) (not (MU-stg-p stg)))
         (implies (BU-stg-p stg) (not (IU-stg-p stg)))
         (implies (MU-stg-p stg) (not (LSU-stg-p stg)))
```

```
         (implies (MU-stg-p stg) (not (BU-stg-p stg)))
         (implies (MU-stg-p stg) (not (IU-stg-p stg))))
  :hints (("goal" :in-theory (enable BU-stg-p IU-stg-p MU-stg-p LSU-stg-p))))

; Wbuf-stg-p and other stages are exclusive.
(defthm wbuf-stg-not-IFU-stg
    (implies (wbuf-stg-p stg)
             (not (IFU-stg-p stg)))
  :hints (("goal" :in-theory (enable wbuf-stg-p IFU-stg-p))))

(defthm wbuf-stg-not-dq-stg
    (implies (wbuf-stg-p stg)
             (not (dq-stg-p stg)))
  :hints (("goal" :in-theory (enable wbuf-stg-p dq-stg-p))))

(defthm wbuf-stg-not-retire-stg
    (implies (wbuf-stg-p stg)
             (not (retire-stg-p stg)))
  :hints (("goal" :in-theory (enable wbuf-stg-p retire-stg-p))))

(defthm wbuf0-stg-p-wbuf1-stg-p-exclusive
    (and (implies (wbuf0-stg-p i) (not (wbuf1-stg-p i)))
         (implies (wbuf1-stg-p i) (not (wbuf0-stg-p i))))
  :hints (("goal" :in-theory (enable wbuf0-stg-p wbuf1-stg-p))))

(defthm wbuf-stg-if-wbuf0-stg
    (implies (wbuf0-stg-p stg) (wbuf-stg-p stg))
  :hints (("goal" :in-theory (enable wbuf0-stg-p wbuf-stg-p))))

(defthm wbuf-stg-if-wbuf1-stg
    (implies (wbuf1-stg-p stg) (wbuf-stg-p stg))
  :hints (("goal" :in-theory (enable wbuf1-stg-p wbuf-stg-p))))

(defthm IFU-stg-p-cons
    (implies (not (equal tag 'IFU))
             (not (IFU-stg-p (cons tag x))))
  :hints (("goal" :in-theory (enable MA-stg-def))))

(defthm DQ-stg-p-cons
    (implies (not (equal tag 'DQ))
             (not (DQ-stg-p (cons tag x))))
  :hints (("goal" :in-theory (enable MA-stg-def))))

(defthm execute-stg-p-cons
    (implies (and (not (equal tag 'IU))
                  (not (equal tag 'BU))
                  (not (equal tag 'LSU))
                  (not (equal tag 'MU)))
             (not (execute-stg-p (cons tag x))))
  :hints (("goal" :in-theory (enable MA-stg-def))))

(defthm complete-stg-p-cons
    (implies (not (equal tag 'complete))
             (not (complete-stg-p (cons tag x))))
  :hints (("goal" :in-theory (enable MA-stg-def))))

(defthm commit-stg-p-cons
    (implies (not (equal tag 'commit))
             (not (commit-stg-p (cons tag x))))
  :hints (("goal" :in-theory (enable MA-stg-def))))

(defthm retire-stg-p-cons
```

```
    (implies (not (equal tag 'retire))
             (not (retire-stg-p (cons tag x))))
  :hints (("goal" :in-theory (enable MA-stg-def))))

(encapsulate nil
(local
(defthm not-IFU-stg-p-if-not-last-INST-help
    (implies (and (in-order-trace-p trace)
                  (tail-p sub trace)
                  (consp sub)
                  (consp (cdr sub)))
             (not (IFU-stg-p (INST-stg (car sub)))))))

;; The instruction at IFU stage is always at the end of a MAETT.
(defthm not-IFU-stg-p-if-not-last-INST
    (implies (and (inv MT MA)
                  (MA-state-p MA) (MAETT-p MT)
                  (subtrace-p trace MT)
                  (consp trace)
                  (consp (cdr trace)))
             (not (IFU-stg-p (INST-stg (car trace)))))
  :hints (("goal" :in-theory (enable inv in-order-dispatch-commit-p
                                     subtrace-p))))
)

; An instruction at IFU stage is always at the end of MAETT.  So any
; instruction j cannot be a member of (cdr trace) if (car trace) is
; IFU-stg-p.  Note: This rule is found to be useful several proofs,
; where a manual hint is required to split the case depending on
; (consp (cdr trace)) or not.
(defthm not-member-of-cdr-if-car-is-IFU-stg
    (implies (and (inv MT MA)
                  (IFU-stg-p (INST-stg (car trace)))
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (consp trace))
             (not (member-equal j (cdr trace))))
  :hints (("goal" :cases ((consp (cdr trace))))))

(encapsulate nil
(local
(defthm IFU-is-last-inst-help
    (implies (and (in-order-trace-p trace)
                  (IFU-stg-p (INST-stg j))
                  (not (equal i j))
                  (member-equal i trace)
                  (member-equal j trace))
             (member-in-order i j trace))
  :hints (("goal" :in-theory (enable member-in-order*)))))

(defthm IFU-is-last-inst
    (implies (and (inv MT MA)
                  (IFU-stg-p (INST-stg j))
                  (not (equal i j))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (INST-in j MT))
             (INST-in-order-p i j MT))
  :hints (("goal" :in-theory (enable INST-in-order-p INST-in
                                     inv in-order-dispatch-commit-p))))
)
```

```
(in-theory (disable IFU-is-last-inst ))

(encapsulate nil
(local
(defthm lower-bound-of-DQ-stg-idx-of-member
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (member-equal j trace)
                  (DQ-stg-p (INST-stg j))
                  (in-order-DQ-trace-p trace idx))
             (<= idx (DQ-stg-idx (INST-stg J))))
  :rule-classes nil))

(local
(defthm DQ-stg-index-monotonic-induct
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (in-order-DQ-trace-p trace idx)
                  (subtrace-p trace MT)
                  (member-in-order i j trace)
                  (DQ-stg-p (INST-stg i))
                  (DQ-stg-p (INST-stg j)))
             (< (DQ-stg-idx (INST-stg i)) (DQ-stg-idx (INST-stg j))))
  :hints (("goal" :in-theory (enable member-in-order*))
          (when-found (binary-+ '1 (DQ-STG-IDX (INST-STG (CAR TRACE))))
              (:use (:instance lower-bound-of-DQ-stg-idx-of-member
                               (idx (+ 1 (DQ-STG-IDX (INST-STG (CAR TRACE)))))
                               (trace (cdr trace)))))))
  :rule-classes nil))

; Dispatch queue works as a FIFO queue.  If instruction i precedes
; instruction j, and they are both in DQ-stg, DQ-stg-idx of i is smaller
; than that of j.
(defthm DQ-stg-index-monotonic
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in-order-p i j MT)
                  (DQ-stg-p (INST-stg i))
                  (DQ-stg-p (INST-stg j)))
             (< (DQ-stg-idx (INST-stg i)) (DQ-stg-idx (INST-stg j))))
  :hints (("goal" :use (:instance DQ-stg-index-monotonic-induct
                                  (trace (MT-trace MT))
                                  (idx 0))
                  :in-theory (enable inv in-order-DQ-p
                                     INST-in-order-p)))
  :rule-classes nil)
)

(defthm DQ0-is-earlier-than-other-DQ
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(DQ 0))
                  (not (equal i j))
                  (DQ-stg-p (INST-stg j))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-p j)
                  (INST-in i MT)
                  (INST-in j MT))
             (INST-in-order-p i j MT))
  :hints (("goal" :use (:instance DQ-stg-index-monotonic
                                  (i j) (j i)))))
```

```
(in-theory (disable DQ0-is-earlier-than-other-DQ))

(encapsulate nil
(local
(defthm non-commit-stg-p-cadr-if-not-committed-p-car-help
    (implies (and (tail-p sub trace)
                  (INST-listp sub)
                  (in-order-trace-p trace)
                  (not (committed-p (car sub))))
             (not (commit-stg-p (INST-stg (cadr sub)))))
  :hints (("goal" :in-theory (enable committed-p)))))

(local
(defthm non-retire-stg-p-cadr-if-not-committed-p-car-help
    (implies (and (tail-p sub trace)
                  (INST-listp sub)
                  (in-order-trace-p trace)
                  (not (committed-p (car sub))))
             (not (retire-stg-p (INST-stg (cadr sub)))))
  :hints (("goal" :in-theory (enable committed-p)))))

; Instructions commit in order.  If car of trace is not committed,
; cadr of trace is not committed either.  A similar lemma follows.
(defthm non-commit-stg-p-cadr-if-not-committed-p-car
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (not (committed-p (car trace))))
             (not (commit-stg-p (INST-stg (cadr trace)))))
  :hints (("goal" :in-theory (enable inv subtrace-p
                                     in-order-dispatch-commit-p))))

(defthm non-retire-stg-p-cadr-if-not-committed-p-car
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (not (committed-p (car trace))))
             (not (retire-stg-p (INST-stg (cadr trace)))))
  :hints (("goal" :in-theory (enable inv subtrace-p
                                     in-order-dispatch-commit-p))))
)

; This lemma shows that the instruction at stage (DQ 0), if it exists,
; is the first non-dispatched instruction in program order.
; Presentation of this lemma is more technical. If instruction i is at
; (DQ 0), and car of trace is a non-dispatched instruction, then i
; cannot be a member of cdr of trace.
(defthm INST-at-DQ-0-is-first-non-dispatched-inst
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(DQ 0))
                  (not (dispatched-p (car trace)))
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace))
             (not (member-equal i (cdr trace))))
  :hints (("goal" :in-theory (enable dispatched-p)
                  :use ((:instance DQ-stg-index-monotonic
                                   (i (car trace))
                                   (j i))
                        (:instance INST-is-at-one-of-the-stages
                                   (i (car trace)))))
          (when-found (IFU-STG-P (INST-STG (CAR TRACE)))
```

```
                    (:cases ((consp (cdr trace))))))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (equal (INST-stg i) '(DQ 0))
                           (member-equal i (cdr trace))
                           (MAETT-p MT) (MA-state-p MA)
                           (subtrace-p trace MT)
                           (INST-listp trace))
                      (dispatched-p (car trace))))))
(in-theory
 (disable (:rewrite INST-at-DQ-0-is-first-non-dispatched-inst . 2)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;; Theory  about  dispatched and committed instructions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deflabel begin-dispatch-commit-inst-stages)

(defthm IFU-not-dispatched-inst
    (implies (IFU-stg-p (INST-stg i)) (not (dispatched-p i))))

(defthm dq-not-dispatched-inst
    (implies (DQ-stg-p (INST-stg i)) (not (dispatched-p i))))

(defthm executed-dispatched-inst
    (implies (execute-stg-p (INST-stg i)) (dispatched-p i)))

(defthm completed-dispatched-inst
    (implies (complete-stg-p (INST-stg i)) (dispatched-p i)))

(defthm commit-dispatched-inst
    (implies (commit-stg-p (INST-stg i)) (dispatched-p i)))

(defthm retired-dispatched-inst
    (implies (retire-stg-p (INST-stg i)) (dispatched-p i)))

(defthm IFU-not-committed-inst
    (implies (IFU-stg-p (INST-stg i)) (not (committed-p i))))

(defthm dq-not-committed-inst
    (implies (DQ-stg-p (INST-stg i)) (not (committed-p i))))

(defthm executed-committed-inst
    (implies (execute-stg-p (INST-stg i)) (not (committed-p i))))

(defthm completed-committed-inst
    (implies (complete-stg-p (INST-stg i)) (not (committed-p i))))

(defthm commit-committed-inst
    (implies (commit-stg-p (INST-stg i)) (committed-p i)))

(defthm retired-committed-inst
    (implies (retire-stg-p (INST-stg i)) (committed-p i)))

(defthm not-committed-p-if-not-commit-retire
    (implies (and (not (commit-stg-p (INST-stg i)))
                  (not (retire-stg-p (INST-stg i))))
             (not (committed-p i)))
  :hints (("goal" :in-theory (enable committed-p))))

(deflabel end-dispatch-commit-inst-stages)
```

```
(deftheory dispatch-commit-inst-stages
    (set-difference-theories
     (universal-theory 'end-dispatch-commit-inst-stages)
     (universal-theory 'begin-dispatch-commit-inst-stages)))

(defthm dispatched-p*
    (implies (INST-p i)
             (equal (dispatched-p i)
                    (not (or (IFU-stg-p (INST-stg i))
                             (DQ-stg-p (INST-stg i))))))
  :hints (("goal" :in-theory (enable dispatched-p)
                  :use (:instance INST-is-at-one-of-the-stages))))

(defthm committed-p*
    (implies (INST-p i)
             (equal (committed-p i)
                    (not (or (IFU-stg-p (INST-stg i))
                             (DQ-stg-p (INST-stg i))
                             (execute-stg-p (INST-stg i))
                             (complete-stg-p (INST-stg i))))))
  :hints (("goal" :in-theory (enable committed-p)
                  :use (:instance INST-is-at-one-of-the-stages))))

(in-theory (disable committed-p* dispatched-p*))

(encapsulate nil
;; These lemmas are locally defined because the derived rules are
;; fired too frequently and slows down the proof process.
(local
(defthm not-member-equal-if-no-dispatched-inst
    (implies (and (dispatched-p i)
                  (no-dispatched-inst-p trace))
             (not (member-equal i trace)))))

(local
(defthm not-member-equal-if-no-commit-inst
    (implies (and (committed-p i)
                  (no-commit-inst-p trace))
             (not (member-equal i trace)))))

(local
(defthm INST-in-order-commit-uncommit-help
    (implies (and (in-order-trace-p trace)
                  (INST-listp trace)
                  (INST-p i) (INST-p j)
                  (member-equal i trace)
                  (member-equal j trace)
                  (committed-p i)
                  (not (committed-p j)))
             (member-in-order i j trace))
  :hints (("goal" :in-theory (enable member-in-order*)
                  :induct t))))

(defthm INST-in-order-commit-uncommit
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (INST-in j MT)
                  (committed-p i)
                  (not (committed-p j)))
             (INST-in-order-p i j MT))
```

```
  :hints (("goal" :in-theory (enable INST-in-order-p INST-in
                                     inv
                                     IN-ORDER-DISPATCH-COMMIT-P))))

(local
(defthm INST-in-order-dispatched-undispatched-help
    (implies (and (in-order-trace-p trace)
                  (INST-listp trace)
                  (member-equal i trace)
                  (member-equal j trace)
                  (dispatched-p i)
                  (not (dispatched-p j)))
             (member-in-order i j trace))
  :hints (("goal" :in-theory (enable member-in-order*)
                  :induct t))))

(defthm INST-in-order-dispatched-undispatched
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (INST-in j MT)
                  (dispatched-p i)
                  (not (dispatched-p j)))
             (INST-in-order-p i j MT))
  :hints (("goal" :in-theory (enable INST-in-order-p INST-in
                                     inv in-order-dispatch-commit-p))))
) ;encapsulate

(in-theory (disable dispatched-p committed-p))

(defthm not-member-equal-cdr-if-car-is-not-commit
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (consp trace)
                  (not (committed-p (car trace)))
                  (committed-p i)
                  (MAETT-p MT) (MA-state-p MA))
             (not (member-equal i (cdr trace))))
  :hints (("goal" :use (:instance INST-IN-ORDER-COMMIT-UNCOMMIT
                                  (i i) (j (car trace)))
                  :in-theory (disable INST-IN-ORDER-COMMIT-UNCOMMIT)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (subtrace-p trace MT)
                           (consp trace)
                           (member-equal i (cdr trace))
                           (committed-p i)
                           (MAETT-p MT) (MA-state-p MA))
                      (committed-p (car trace))))))

(in-theory
 (disable (:rewrite not-member-equal-cdr-if-car-is-not-commit . 2)))

(defthm not-member-equal-cdr-if-car-is-not-dispatched
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (consp trace)
                  (not (dispatched-p (car trace)))
                  (dispatched-p i)
                  (MAETT-p MT) (MA-state-p MA))
```

```
            (not (member-equal i (cdr trace))))
  :hints (("goal" :use (:instance inst-in-order-dispatched-undispatched
                                  (i i) (j (car trace)))
                  :in-theory (disable inst-in-order-dispatched-undispatched)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (subtrace-p trace MT) (INST-listp trace)
                           (consp trace)
                           (member-equal i (cdr trace))
                           (dispatched-p i)
                           (MAETT-p MT) (MA-state-p MA))
                      (dispatched-p (car trace))))))

(in-theory
 (disable (:rewrite not-member-equal-cdr-if-car-is-not-dispatched . 2)))

(encapsulate nil
(local
 (defthm inst-of-tag-is-dispatched-help
     (implies (uniq-inst-of-tag-in-trace rix trace)
              (dispatched-p (inst-of-tag-in-trace rix trace)))
   :hints (("goal" :in-theory (enable dispatched-p )))))

(defthm inst-of-tag-is-dispatched
    (implies (uniq-inst-of-tag rix MT)
             (dispatched-p (inst-of-tag rix MT)))
  :hints (("goal" :in-theory (enable inst-of-tag uniq-inst-of-tag)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (uniq-inst-of-tag rix MT)
                      (not (IFU-stg-p (INST-stg (inst-of-tag rix MT))))))
   (:rewrite :corollary
             (implies (uniq-inst-of-tag rix MT)
                      (not (DQ-stg-p (INST-stg (inst-of-tag rix MT))))))))
)

(encapsulate nil
(local
 (defthm inst-of-tag-is-not-committed-help
     (implies (uniq-inst-of-tag-in-trace rix trace)
              (not (committed-p (inst-of-tag-in-trace rix trace))))
   :hints (("goal" :in-theory (enable committed-p )))))

(defthm inst-of-tag-is-not-committed
    (implies (uniq-inst-of-tag rix MT)
             (not (committed-p (inst-of-tag rix MT))))
  :hints (("goal" :in-theory (enable inst-of-tag uniq-inst-of-tag)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (uniq-inst-of-tag rix MT)
                      (not (retire-stg-p (INST-stg (inst-of-tag rix MT))))))
   (:rewrite :corollary
             (implies (uniq-inst-of-tag rix MT)
                      (not (commit-stg-p (INST-stg (inst-of-tag rix MT))))))))

)

;; End of the theory  about  dispatched and committed instructions
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Theories about init-MT
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(encapsulate nil
 (local (in-theory (enable init-MT)))

(defthm MT-init-ISA-init-MT
    (equal (MT-init-ISA (init-MT MA))
           (proj MA)))

(defthm MT-new-ID-init-MT
    (equal (MT-new-ID (init-MT MA)) 0))

(defthm MT-dq-len-init-MT
    (equal (MT-dq-len (init-MT MA)) 0))

(defthm MT-rob-head-init-MT
    (equal (MT-rob-head (init-MT MA)) (ROB-head (MA-rob MA))))

(defthm MT-rob-tail-init-MT
    (equal (MT-rob-tail (init-MT MA)) (ROB-tail (MA-rob MA))))

(defthm MT-trace-init-MT
    (equal (MT-trace (init-MT MA)) nil))
)

(defthm MT-init-ISA-MT-step
    (equal (MT-init-ISA (MT-step MT MA sigs))
           (MT-init-ISA MT))
  :hints (("Goal" :in-theory (enable MT-step))))

(defthm MT-init-ISA-MT-stepn
    (equal (MT-init-ISA (MT-stepn MT MA sigs-lst n))
           (MT-init-ISA MT)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Lemmas about microarchitecture
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defthm not-fetch-inst-if-ex-intr
    (implies (b1p (ex-intr? MA sigs))
             (equal (fetch-inst? MA sigs) 0))
  :hints (("Goal" :in-theory (enable fetch-inst? lift-b-ops flush-all?
                                     b1p-bit-rewriter))))

(defthm flush-all-fetch-inst-exclusive
    (implies (b1p (fetch-inst? MA sigs))
             (equal (flush-all? MA sigs) 0))
  :Hints (("goal" :in-theory (enable fetch-inst? lift-b-ops
                                     equal-b1p-converter))))

(defthm not-fetch-inst-if-dq-full
    (implies (and (b1p (IFU-valid? (MA-IFU MA)))
                  (b1p (DQ-full? (MA-dq MA))))
             (equal (fetch-inst? MA sigs) 0))
  :hints (("Goal" :in-theory (enable fetch-inst? lift-b-ops
                                     equal-b1p-converter))))

; The proof of dispatch-to-IU-no-unit-exclusive If dispatch-to-IU is
; true, there should be an instruction at DE0.  This instruction
; should satisfy INST-inv.  We can prove that dispatch-to-IU? and
; dispatch-no-unit? are not 1 simultaneously in a reachable state.
```

```
(encapsulate nil
(local
(defthm dispatch-to-IU-no-unit-exclusive-help
    (implies (and (inv MT MA)
                  (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))
                  (b1p (logbit 0 (cntlv-exunit
                                  (DE-cntlv (DQ-DE0 (MA-DQ MA)))))))
             (not (b1p (logbit 4 (cntlv-exunit
                                  (DE-cntlv (DQ-DE0 (MA-DQ MA))))))))
    :hints (("goal" :in-theory (enable consistent-cntlv-p
                                       inv CONSISTENT-MA-P
                                       CONSISTENT-DQ-CNTLV-P
                                       lift-b-ops)))))


(defthm dispatch-to-IU-no-unit-exclusive
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-IU? MA)))
             (not (b1p (dispatch-no-unit? MA))))
  :hints (("goal" :in-theory (e/d (dispatch-to-IU? dispatch-no-unit?
                                   DQ-ready-to-IU? DQ-ready-no-unit?
                                   dispatch-inst?
                                   lift-b-ops) ))))
)

(encapsulate nil
(local
(defthm dispatch-to-MU-no-unit-exclusive-help
    (implies (and (inv MT MA)
                  (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))
                  (b1p (logbit 1 (cntlv-exunit (DE-cntlv
                                                (DQ-DE0 (MA-DQ MA)))))))
             (not (b1p (logbit 4 (cntlv-exunit (DE-cntlv
                                                (DQ-DE0 (MA-DQ MA))))))))
    :hints (("goal" :in-theory (enable consistent-cntlv-p
                                       inv CONSISTENT-MA-P
                                       CONSISTENT-DQ-CNTLV-P
                                       lift-b-ops)))))


(defthm dispatch-to-MU-no-unit-exclusive
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-MU? MA)))
             (not (b1p (dispatch-no-unit? MA))))
  :hints (("goal" :in-theory (e/d (dispatch-to-MU? dispatch-no-unit?
                                   DQ-ready-to-MU? DQ-ready-no-unit?
                                   dispatch-inst?
                                   lift-b-ops) ))))
)

(encapsulate nil
(local
  (defthm dispatch-to-LSU-no-unit-exclusive-help
      (implies (and (inv MT MA)
                    (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))
                    (b1p (logbit 2 (cntlv-exunit (DE-cntlv (DQ-DE0 (MA-DQ MA)))))))
               (not (b1p (logbit 4 (cntlv-exunit (DE-cntlv (DQ-DE0 (MA-DQ MA))))))))

    :hints (("goal" :in-theory (enable consistent-cntlv-p
                                       inv CONSISTENT-MA-P
                                       CONSISTENT-DQ-CNTLV-P
                                       lift-b-ops)))))
```

```
(defthm dispatch-to-LSU-no-unit-exclusive
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-LSU? MA)))
             (not (b1p (dispatch-no-unit? MA))))
  :hints (("goal" :in-theory (e/d (dispatch-to-LSU? dispatch-no-unit?
                                   DQ-ready-to-LSU? DQ-ready-no-unit?
                                   dispatch-inst?
                                   lift-b-ops) ))))
)

(encapsulate nil
(local
(defthm dispatch-to-BU-no-unit-exclusive-help
    (implies (and (inv MT MA)
                  (b1p (DE-valid? (DQ-DEO (MA-DQ MA))))
                  (b1p (logbit 3 (cntlv-exunit (DE-cntlv
                                                (DQ-DEO (MA-DQ MA)))))))
             (not (b1p (logbit 4 (cntlv-exunit (DE-cntlv
                                                (DQ-DEO (MA-DQ MA))))))))
    :hints (("goal" :in-theory (enable consistent-cntlv-p
                                       inv CONSISTENT-MA-P
                                       CONSISTENT-DQ-CNTLV-P
                                       lift-b-ops)))))


(defthm dispatch-to-BU-no-unit-exclusive
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-BU? MA)))
             (not (b1p (dispatch-no-unit? MA))))
  :hints (("goal" :in-theory (e/d (dispatch-to-BU? dispatch-no-unit?
                                   DQ-ready-to-BU? DQ-ready-no-unit?
                                   dispatch-inst?
                                   lift-b-ops) ))))
)

(encapsulate nil
(local
(defthm dispatch-to-MU-IU-exclusive-help
    (implies (and (inv MT MA)
                  (b1p (DE-valid? (DQ-DEO (MA-DQ MA))))
                  (b1p (logbit 1 (cntlv-exunit (DE-cntlv
                                                (DQ-DEO (MA-DQ MA)))))))
             (not (b1p (logbit 0 (cntlv-exunit (DE-cntlv
                                                (DQ-DEO (MA-DQ MA))))))))

    :hints (("goal" :in-theory (enable consistent-cntlv-p
                                       inv CONSISTENT-MA-P
                                       CONSISTENT-DQ-CNTLV-P
                                       lift-b-ops)))))


(defthm dispatch-to-MU-IU-exclusive
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-MU? MA)))
             (not (b1p (dispatch-to-IU? MA))))
  :hints (("goal" :in-theory (e/d (dispatch-to-MU? dispatch-to-IU?
                                   DQ-ready-to-MU? DQ-ready-to-IU?
                                   dispatch-inst?
                                   lift-b-ops) ))))
)
```

```
(encapsulate nil
(local
(defthm dispatch-to-LSU-IU-exclusive-help
    (implies (and (inv MT MA)
                  (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))
                  (b1p (logbit 2 (cntlv-exunit (DE-cntlv
                                                (DQ-DE0 (MA-DQ MA)))))))
             (not (b1p (logbit 0 (cntlv-exunit (DE-cntlv
                                                (DQ-DE0 (MA-DQ MA))))))))
      :hints (("goal" :in-theory (enable consistent-cntlv-p
                                         inv CONSISTENT-MA-P
                                         CONSISTENT-DQ-CNTLV-P
                                         lift-b-ops)))))


(defthm dispatch-to-LSU-IU-exclusive
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-LSU? MA)))
             (not (b1p (dispatch-to-IU? MA))))
  :hints (("goal" :in-theory (e/d (dispatch-to-LSU? dispatch-to-IU?
                                   DQ-ready-to-LSU? DQ-ready-to-IU?
                                   dispatch-inst?
                                   lift-b-ops) ))))
)

(encapsulate nil
(local
(defthm dispatch-to-BU-IU-exclusive-help
    (implies (and (inv MT MA)
                  (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))
                  (b1p (logbit 3 (cntlv-exunit (DE-cntlv
                                                (DQ-DE0 (MA-DQ MA)))))))
             (not (b1p (logbit 0 (cntlv-exunit (DE-cntlv
                                                (DQ-DE0 (MA-DQ MA))))))))
    :hints (("goal" :in-theory (enable consistent-cntlv-p
                                       inv CONSISTENT-MA-P
                                       CONSISTENT-DQ-CNTLV-P
                                       lift-b-ops)))))


(defthm dispatch-to-BU-IU-exclusive
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-BU? MA)))
             (not (b1p (dispatch-to-IU? MA))))
  :hints (("goal" :in-theory (e/d (dispatch-to-BU? dispatch-to-IU?
                                   DQ-ready-to-BU? DQ-ready-to-IU?
                                   dispatch-inst?
                                   lift-b-ops) ))))
)

(encapsulate nil
(local
(defthm dispatch-to-LSU-MU-exclusive-help
    (implies (and (inv MT MA)
                  (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))
                  (b1p (logbit 2 (cntlv-exunit (DE-cntlv
                                                (DQ-DE0 (MA-DQ MA)))))))
             (not (b1p (logbit 1 (cntlv-exunit (DE-cntlv
                                                (DQ-DE0 (MA-DQ MA))))))))
    :hints (("goal" :in-theory (enable consistent-cntlv-p
                                       inv CONSISTENT-MA-P
```

```
                                    CONSISTENT-DQ-CNTLV-P
                                    lift-b-ops)))))


(defthm dispatch-to-LSU-MU-exclusive
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-LSU? MA)))
             (not (b1p (dispatch-to-MU? MA))))
  :hints (("goal" :in-theory (e/d (dispatch-to-LSU? dispatch-to-MU?
                                   DQ-ready-to-LSU? DQ-ready-to-MU?
                                   dispatch-inst?
                                   lift-b-ops) ))))
)

(encapsulate nil
(local
(defthm dispatch-to-BU-MU-exclusive-help
    (implies (and (inv MT MA)
                  (b1p (DE-valid? (DQ-DEO (MA-DQ MA))))
                  (b1p (logbit 3 (cntlv-exunit (DE-cntlv
                                                (DQ-DEO (MA-DQ MA)))))))
             (not (b1p (logbit 1 (cntlv-exunit (DE-cntlv
                                                (DQ-DEO (MA-DQ MA))))))))
    :hints (("goal" :in-theory (enable consistent-cntlv-p
                                    inv CONSISTENT-MA-P
                                    CONSISTENT-DQ-CNTLV-P
                                    lift-b-ops)))))


(defthm dispatch-to-BU-MU-exclusive
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-BU? MA)))
             (not (b1p (dispatch-to-MU? MA))))
  :hints (("goal" :in-theory (e/d (dispatch-to-BU? dispatch-to-MU?
                                   DQ-ready-to-BU? DQ-ready-to-MU?
                                   dispatch-inst?
                                   lift-b-ops) ))))
)

(encapsulate nil
(local
  (defthm dispatch-to-BU-LSU-exclusive-help
      (implies (and (inv MT MA)
                    (b1p (DE-valid? (DQ-DEO (MA-DQ MA))))
                    (b1p (logbit 3 (cntlv-exunit (DE-cntlv (DQ-DEO (MA-DQ MA)))))))
               (not (b1p (logbit 2 (cntlv-exunit (DE-cntlv (DQ-DEO (MA-DQ MA))))))))

    :hints (("goal" :in-theory (enable consistent-cntlv-p
                                    inv CONSISTENT-MA-P
                                    CONSISTENT-DQ-CNTLV-P
                                    lift-b-ops)))))


(defthm dispatch-to-BU-LSU-exclusive
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-BU? MA)))
             (not (b1p (dispatch-to-LSU? MA))))
  :hints (("goal" :in-theory (e/d (dispatch-to-BU? dispatch-to-LSU?
                                   DQ-ready-to-BU? DQ-ready-to-LSU?
                                   dispatch-inst?
                                   lift-b-ops) ))))
)
```

```
(defthm DE-valid-consistent
    (and (implies (and (inv MT MA)
                       (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))))
                  (not (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))))
         (implies (and (inv MT MA)
                       (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))))
                  (not (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))))
         (implies (and (inv MT MA)
                       (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))))
                  (not (b1p (DE-valid? (DQ-DE3 (MA-DQ MA))))))
         (implies (and (inv MT MA)
                       (b1p (DE-valid? (DQ-DE1 (MA-DQ MA)))))
                  (b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))
         (implies (and (inv MT MA)
                       (not (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))))
                  (not (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))))
         (implies (and (inv MT MA)
                       (not (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))))
                  (not (b1p (DE-valid? (DQ-DE3 (MA-DQ MA))))))
         (implies (and (inv MT MA)
                       (b1p (DE-valid? (DQ-DE2 (MA-DQ MA)))))
                  (b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))
         (implies (and (inv MT MA)
                       (b1p (DE-valid? (DQ-DE2 (MA-DQ MA)))))
                  (b1p (DE-valid? (DQ-DE1 (MA-DQ MA)))))
         (implies (and (inv MT MA)
                       (not (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))))
                  (not (b1p (DE-valid? (DQ-DE3 (MA-DQ MA))))))
         (implies (and (inv MT MA)
                       (b1p (DE-valid? (DQ-DE3 (MA-DQ MA)))))
                  (b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))
         (implies (and (inv MT MA)
                       (b1p (DE-valid? (DQ-DE3 (MA-DQ MA)))))
                  (b1p (DE-valid? (DQ-DE1 (MA-DQ MA)))))
         (implies (and (inv MT MA)
                       (b1p (DE-valid? (DQ-DE3 (MA-DQ MA)))))
                  (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))))
  :hints (("goal" :in-theory (enable inv misc-inv
                                     correct-entries-in-DQ-p))))

(defthm LSU-wbuf0-valid-if-LSU-wbuf1-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
             (equal (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))) 1))
  :hints (("goal" :in-theory (enable inv consistent-MA-p
                                     consistent-LSU-p
                                     equal-b1p-converter))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Lemmas about stages after step-INST
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deflabel begin-step-INST-opener)
(in-theory (enable step-INST))
(defthm step-INST-IFU-INST
    (implies (IFU-stg-p (INST-stg i))
             (equal (step-INST i MT MA sigs)
                    (step-INST-IFU I MT MA sigs))))

(defthm step-INST-DQ-INST
    (implies (DQ-stg-p (INST-stg i))
```

```
            (equal (step-INST i MT MA sigs)
                   (step-INST-DQ I MT MA sigs))))

(defthm step-INST-execute-INST
    (implies (execute-stg-p (INST-stg i))
             (equal (step-INST i MT MA sigs)
                    (step-INST-execute I MA sigs))))

(defthm step-INST-complete-INST
    (implies (complete-stg-p (INST-stg i))
             (equal (step-INST i MT MA sigs)
                    (step-INST-complete I MA sigs))))

(defthm step-INST-commit-INST
    (implies (commit-stg-p (INST-stg i))
             (equal (step-INST i MT MA sigs)
                    (step-INST-commit I MA sigs))))

(in-theory (disable step-INST))
(deflabel end-step-INST-opener)
(deftheory step-INST-opener
    (set-difference-theories (universal-theory 'end-step-INST-opener)
                             (universal-theory 'begin-step-INST-opener)))
(in-theory (disable step-INST-opener))

(defthm INST-stg-exintr-INST
    (equal (INST-stg (exintr-INST MT ISA smc)) '(retire))
  :hints (("goal" :in-theory (enable exintr-INST))))

(defthm INST-stg-fetched-inst
    (equal (INST-stg (fetched-inst MT ISA spc smc)) '(IFU))
  :hints (("goal" :in-theory (enable fetched-inst))))

(defthm not-IFU-stg-step-INST-if-not-IFU
    (implies (not (IFU-stg-p (INST-stg i)))
             (not (IFU-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("goal" :in-theory (enable MT-def))))

(defthm not-IFU-stg-step-INST-if-not-IFU-2
    (implies (not (IFU-stg-p (INST-stg i)))
             (not (equal (INST-stg (step-INST i MT MA sigs)) '(IFU))))
  :hints (("goal" :in-theory (enable IFU-stg-p)
                  :use (:instance not-IFU-stg-step-INST-if-not-IFU))))

(defthm not-DQ-stg-step-INST-if-not-IFU
    (implies (and (not (IFU-stg-p (INST-stg i)))
                  (not (DQ-stg-p (INST-stg i))))
             (not (DQ-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("goal" :in-theory (enable MT-def))))

(defthm not-DQ-stg-step-INST-if-not-IFU-2
    (implies (and (INST-p i)
                  (not (IFU-stg-p (INST-stg i)))
                  (not (DQ-stg-p (INST-stg i))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         (list 'DQ idx))))
  :hints (("goal" :in-theory (enable MT-def MA-stg-def)
                  :use (inst-is-at-one-of-the-stages))))

(defthm not-execute-stg-step-INST-if-not-DQ
    (implies (and (not (DQ-stg-p (INST-stg i)))
                  (not (execute-stg-p (INST-stg i))))
```

```
            (not (execute-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("goal" :in-theory (enable MT-def))))

(defthm not-IU-stg-step-INST-if-not-DQ-2
    (implies (and (INST-p i)
                  (not (DQ-stg-p (INST-stg i)))
                  (not (execute-stg-p (INST-stg i))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         (cons 'IU trailer))))
  :hints (("goal" :in-theory (enable MT-def MA-stg-def)
                  :use (inst-is-at-one-of-the-stages))))

(defthm not-BU-stg-step-INST-if-not-DQ-2
    (implies (and (INST-p i)
                  (not (DQ-stg-p (INST-stg i)))
                  (not (execute-stg-p (INST-stg i))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         (cons 'BU trailer))))
  :hints (("goal" :in-theory (enable MT-def MA-stg-def)
                  :use (inst-is-at-one-of-the-stages))))

(defthm not-MU-stg-step-INST-if-not-DQ-2
    (implies (and (INST-p i)
                  (not (DQ-stg-p (INST-stg i)))
                  (not (execute-stg-p (INST-stg i))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         (cons 'MU trailer))))
  :hints (("goal" :in-theory (enable MT-def MA-stg-def)
                  :use (inst-is-at-one-of-the-stages))))

(defthm not-LSU-stg-step-INST-if-not-DQ-2
    (implies (and (INST-p i)
                  (not (DQ-stg-p (INST-stg i)))
                  (not (execute-stg-p (INST-stg i))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         (cons 'LSU trailer))))
  :hints (("goal" :in-theory (enable MT-def MA-stg-def)
                  :use (inst-is-at-one-of-the-stages))))

(defthm not-complete-stg-step-INST-if-not-execute-or-DQ
    (implies (and (not (DQ-stg-p (INST-stg i)))
                  (not (execute-stg-p (INST-stg i)))
                  (not (complete-stg-p (INST-stg i))))
             (not (complete-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("goal" :in-theory (enable MT-def))))

(defthm not-complete-stg-step-INST-if-not-execute-or-DQ-2
    (implies (and (INST-p i)
                  (not (DQ-stg-p (INST-stg i)))
                  (not (execute-stg-p (INST-stg i)))
                  (not (complete-stg-p (INST-stg i))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         (cons 'complete trailer))))
  :hints (("goal" :in-theory (enable MT-def MA-stg-def)
                  :use (inst-is-at-one-of-the-stages))))

(defthm not-commit-stg-step-INST-if-not-commit-or-complete
    (implies (and (not (complete-stg-p (INST-stg i)))
                  (not (commit-stg-p (INST-stg i))))
             (not (commit-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("goal" :in-theory (enable MT-def))))
```

```
(defthm not-commit-stg-step-INST-if-not-commit-or-complete-2
    (implies (and (INST-p i)
                  (not (complete-stg-p (INST-stg i)))
                  (not (commit-stg-p (INST-stg i))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         (cons 'commit trailer))))
  :hints (("goal" :in-theory (enable MT-def MA-stg-def)
                  :use (inst-is-at-one-of-the-stages))))

(defthm not-retire-stg-step-INST-if-not-commit-or-complete
    (implies (and (not (complete-stg-p (INST-stg i)))
                  (not (commit-stg-p (INST-stg i)))
                  (not (retire-stg-p (INST-stg i))))
             (not (retire-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("goal" :in-theory (enable MT-def))))

(defthm not-retire-stg-step-INST-if-not-commit-or-complete-2
    (implies (and (INST-p i)
                  (not (complete-stg-p (INST-stg i)))
                  (not (commit-stg-p (INST-stg i)))
                  (not (retire-stg-p (INST-stg i))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(retire))))
  :hints (("goal" :in-theory (enable MT-def MA-stg-def)
                  :use (inst-is-at-one-of-the-stages))))

(defthm IFU-or-DQ-stg-step-INST-if-IFU
    (implies (IFU-stg-p (INST-stg i))
             (or (IFU-stg-p (INST-stg (step-INST i MT MA sigs)))
                 (DQ-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("goal" :in-theory (enable MT-def DQ-stg-p)))
  :rule-classes
  ((:rewrite :corollary
     (implies (and (IFU-stg-p (INST-stg i))
                   (not (IFU-stg-p (INST-stg (step-INST i MT MA sigs)))))
              (DQ-stg-p (INST-stg (step-INST i MT MA sigs)))))))

(defthm execute-or-complete-stg-step-INST-if-DQ
    (implies (DQ-stg-p (INST-stg i))
             (or (DQ-stg-p (INST-stg (step-INST i MT MA sigs)))
                 (execute-stg-p (INST-stg (step-INST i MT MA sigs)))
                 (complete-stg-p (INST-stg (step-INST i MT MA sigs)))))
    :hints (("goal" :in-theory (enable MT-def DQ-stg-p)))
    :rule-classes
    ((:rewrite :corollary
       (implies (and (DQ-stg-p (INST-stg i))
                     (not (DQ-stg-p (INST-stg (step-INST i MT MA sigs))))
                     (not (execute-stg-p (INST-stg (step-INST i MT MA sigs)))))
                (complete-stg-p (INST-stg (step-INST i MT MA sigs)))))))

(defthm execute-or-complete-stg-step-INST-if-execute
    (implies (execute-stg-p (INST-stg i))
             (or (execute-stg-p (INST-stg (step-INST i MT MA sigs)))
                 (complete-stg-p (INST-stg (step-INST i MT MA sigs)))))
    :hints (("goal" :in-theory (enable MT-def)))
    :rule-classes
    ((:rewrite :corollary
       (implies (and (execute-stg-p (INST-stg i))
                     (not (execute-stg-p (INST-stg (step-INST i MT MA sigs)))))
                (complete-stg-p (INST-stg (step-INST i MT MA sigs)))))))

(defthm commit-or-retire-stg-step-INST-if-complete
```

```
    (implies (complete-stg-p (INST-stg i))
             (or (complete-stg-p (INST-stg (step-INST i MT MA sigs)))
                 (commit-stg-p (INST-stg (step-INST i MT MA sigs)))
                 (retire-stg-p (INST-stg (step-INST i MT MA sigs)))))
    :hints (("goal" :in-theory (enable MT-def)))
    :rule-classes
    ((:rewrite :corollary
       (implies (and (complete-stg-p (INST-stg i))
                     (not (complete-stg-p (INST-stg (step-INST i MT MA sigs))))
                     (not (commit-stg-p (INST-stg (step-INST i MT MA sigs)))))
                (retire-stg-p (INST-stg (step-INST i MT MA sigs)))))))

(defthm retire-or-commit-stg-step-INST-if-commit
    (implies (commit-stg-p (INST-stg i))
             (or (commit-stg-p (INST-stg (step-INST i MT MA sigs)))
                 (retire-stg-p (INST-stg (step-INST i MT MA sigs)))))
    :hints (("goal" :in-theory (enable MT-def)))
    :rule-classes
    ((:rewrite :corollary
       (implies (and (commit-stg-p (INST-stg i))
                     (not (commit-stg-p (INST-stg (step-INST i MT MA sigs)))))
                (retire-stg-p (INST-stg (step-INST i MT MA sigs)))))))

(defthm retire-stg-step-INST-if-retire
    (implies (retire-stg-p (INST-stg i))
             (retire-stg-p (INST-stg (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (enable MT-def))))

(defthm RS-stg-p-step-INST-if-DQ-stg-p
    (implies (and (DQ-stg-p (INST-stg i))
                  (execute-stg-p (INST-stg (step-INST i MT MA sigs))))
             (RS-stg-p (INST-stg (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (enable lift-b-ops STEP-INST-DQ
                                     step-INST-dq-inst
                                     dispatch-inst))))

(defthm RS-stg-p-step-INST-if-DQ-stg-p-col1
    (implies (and (DQ-stg-p (INST-stg i))
                  (not (RS-stg-p (cons 'MU trailer))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         (cons 'MU trailer))))
  :hints (("goal" :in-theory (enable lift-b-ops STEP-INST-DQ
                                     step-INST-dq-inst
                                     dq-stg-p
                                     dispatch-inst))))

(defthm RS-stg-p-step-INST-if-DQ-stg-p-col2
    (implies (and (DQ-stg-p (INST-stg i))
                  (not (RS-stg-p (cons 'LSU trailer))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         (cons 'LSU trailer))))
  :hints (("goal" :in-theory (enable lift-b-ops STEP-INST-DQ
                                     step-INST-dq-inst
                                     dq-stg-p
                                     dispatch-inst))))

(defthm WBUF-stg-p-step-DQ-INST
    (implies (DQ-stg-p (INST-stg i))
             (not (wbuf-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("goal" :in-theory (enable step-inst-dq-inst
                                     dq-stg-p wbuf-stg-p
                                     step-inst-low-level-functions)))
```

```
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (and (wbuf-stg-p stg)
                           (DQ-stg-p (INST-stg i)))
                      (not (equal (INST-stg (step-INST i MT MA sigs)) stg)))
             :hints (("goal" :in-theory (enable wbuf-stg-p))))))

;; More specific lemmas about stages and step-INST
(defthm INST-stg-step-IFU-inst-if-DQ-full
    (implies (IFU-stg-p (INST-stg i))
             (equal (INST-stg (step-INST i MT MA sigs))
                    (b-if (DQ-full? (MA-DQ MA))
                          '(IFU)
                          (NEW-dq-stage MT MA))))
  :hints (("goal" :in-theory (enable MT-def IFU-stg-p))))

(defthm INST-stg-step-DQ-inst-if-not-dispatch
    (implies (and (not (b1p (dispatch-inst? MA)))
                  (DQ-stg-p (inst-stg i)))
             (equal (inst-stg (step-INST i MT MA sigs))
                    (inst-stg i)))
  :hints (("Goal" :in-theory (enable MT-def))))

(defthm INST-stg-step-DQ-inst-if-dispatch-1
    (implies (and (b1p (dispatch-inst? MA))
                  (dq-stg-p (inst-stg i))
                  (not (zp (DQ-stg-idx (INST-stg i)))))
             (equal (inst-stg (step-INST i MT MA sigs))
                    (list 'dq (nfix (1- (DQ-stg-idx (inst-stg i)))))))
  :hints (("goal" :in-theory (enable MT-def))))

(defthm INST-stg-step-DQ-inst-if-dispatch-2
    (implies (and (b1p (dispatch-inst? MA))
                  (equal (inst-stg i) '(DQ 0)))
             (not (equal (inst-stg (step-INST i MT MA sigs)) (list 'DQ idx))))
  :hints (("Goal" :in-theory (enable MT-def))))

(defthm complete-stg-p-step-inst-if-LSU-lch
    (implies (and (equal (INST-stg i) '(LSU lch))
                  (not (equal (INST-stg (step-INST i MT MA sigs))
                              '(LSU lch))))
             (complete-stg-p (INST-stg (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (enable MT-def))))

(defthm complete-stg-p-step-inst-if-LSU-wbuf0-lch
    (implies (and (equal (INST-stg i) '(LSU wbuf0 lch))
                  (not (equal (INST-stg (step-INST i MT MA sigs))
                              '(LSU wbuf0 lch))))
             (complete-stg-p (INST-stg (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (enable MT-def))))

(defthm complete-stg-p-step-inst-if-LSU-wbuf1-lch
    (implies (and (equal (INST-stg i) '(LSU wbuf1 lch))
                  (not (equal (INST-stg (step-INST i MT MA sigs))
                              '(LSU wbuf0 lch)))
                  (not (equal (INST-stg (step-INST i MT MA sigs))
                              '(LSU wbuf1 lch))))
             (complete-stg-p (INST-stg (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (enable MT-def))))

(defthm LSU-stages-direct-reachable-to-complete
```

```
    (implies (and (LSU-stg-p (INST-stg i))
                  (not (or (equal (INST-stg i) '(LSU lch))
                           (equal (INST-stg i) '(LSU wbuf0 lch))
                           (equal (INST-stg i) '(LSU wbuf1 lch)))))
             (not (complete-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("goal" :in-theory (enable LSU-stg-p step-INST
                                     step-INST-execute
                                     step-INST-low-level-functions))))

; An instruction dispatch is never undone.
(defthm dispatch-inst-p-step-inst-if-dispatch
    (implies (dispatched-p i)
             (dispatched-p (step-INST i MT MA sigs)))
  :hints (("goal" :in-theory (enable step-inst
                                     step-INST-low-level-functions))))

(defthm dispatched-inst-step-inst-if-not-dispatch-inst
    (implies (and (INST-p i) (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (not (b1p (dispatch-inst? MA)))
                  (not (dispatched-p i)))
             (not (dispatched-p (step-INST i MT MA sigs))))
  :hints (("Goal" :in-theory (enable dispatched-p*))))

(defthm dispatched-p-step-INST-if-dispatch-inst
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (b1p (dispatch-inst? MA))
                  (equal (INST-stg i) '(DQ 0))
                  (MAETT-p MT) (MA-state-p MA))
             (dispatched-p (step-INST i MT MA sigs)))
  :hints (("Goal" :in-theory (enable dispatched-p
                                     step-inst-dq-inst
                                     step-inst-low-level-functions ))))

; An instruction is once committed, it will be in the future forever.
(defthm committed-p-step-inst-if-commit
    (implies (committed-p i)
             (committed-p (step-INST i MT MA sigs)))
  :hints (("goal" :in-theory (enable step-inst
                                     step-INST-low-level-functions))))

(deflabel begin-inst-stg-step-inst)

; Stage inference rules for instructions at (DQ 0)
; Following Lemmas show to which stage an instruction is dispatched.
(defthm INST-stg-step-INST-if-dispatch-no-unit
    (implies (and (equal (INST-stg i) '(DQ 0))
                  (b1p (dispatch-no-unit? MA)))
             (equal (INST-stg (step-INST i MT MA sigs))
                    '(complete)))
  :hints (("goal" :in-theory (enable step-INST step-INST-dq
                                     dispatch-inst? lift-b-ops
                                     dispatch-inst))))

(defthm INST-stg-step-INST-if-dispatch-IU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(DQ 0))
                  (b1p (dispatch-to-IU? MA)))
             (equal (INST-stg (step-INST i MT MA sigs))
                    (b-if (select-IU-RS0? (MA-IU MA))
```

```
                        '(IU RS0)
                        '(IU RS1))))
  :hints (("goal" :in-theory (enable step-INST step-INST-dq
                                     dispatch-inst? lift-b-ops
                                     dispatch-inst))))

(defthm INST-stg-step-INST-if-dispatch-MU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(DQ 0))
                  (b1p (dispatch-to-MU? MA)))
             (equal (INST-stg (step-INST i MT MA sigs))
                    (b-if (select-MU-RS0? (MA-MU MA))
                          '(MU RS0)
                          '(MU RS1))))
  :hints (("goal" :in-theory (enable step-INST step-INST-dq
                                     dispatch-inst? lift-b-ops
                                     dispatch-inst))))

(defthm INST-stg-step-INST-if-dispatch-BU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(DQ 0))
                  (b1p (dispatch-to-BU? MA)))
             (equal (INST-stg (step-INST i MT MA sigs))
                    (b-if (select-BU-RS0? (MA-BU MA))
                          '(BU RS0)
                          '(BU RS1))))
  :hints (("goal" :in-theory (enable step-INST step-INST-dq
                                     dispatch-inst? lift-b-ops
                                     dispatch-inst))))

(defthm INST-stg-step-INST-if-dispatch-LSU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(DQ 0))
                  (b1p (dispatch-to-LSU? MA)))
             (equal (INST-stg (step-INST i MT MA sigs))
                    (b-if (select-LSU-RS0? (MA-LSU MA))
                          '(LSU RS0)
                          '(LSU RS1))))
  :hints (("goal" :in-theory (enable step-INST step-INST-dq
                                     dispatch-inst? lift-b-ops
                                     dispatch-inst))))

(defthm INST-stg-step-INST-IU-RS0
    (implies (equal (INST-stg i) '(IU RS0))
             (equal (INST-stg (step-INST i MT MA sigs))
                    (if (b1p (issue-IU-RS0? (MA-IU MA) MA))
                        '(complete) '(IU RS0))))
  :hints (("goal" :in-theory (enable step-INST-opener  step-INST-execute
                                     step-INST-IU))))

(defthm INST-stg-step-INST-IU-RS1
    (implies (equal (INST-stg i) '(IU RS1))
             (equal (INST-stg (step-INST i MT MA sigs))
                    (if (b1p (issue-IU-RS1? (MA-IU MA) MA))
                        '(complete) '(IU RS1))))
  :hints (("goal" :in-theory (enable step-INST-opener  step-INST-execute
                                     step-INST-IU))))

(defthm INST-stg-step-INST-MU-RS0
```

```
    (implies (equal (INST-stg i) '(MU RS0))
             (equal (INST-stg (step-INST i MT MA sigs))
                    (if (b1p (issue-MU-RS0? (MA-MU MA) MA))
                        '(MU lch1) '(MU RS0))))
  :hints (("goal" :in-theory (enable step-INST-opener  step-INST-execute
                                     step-INST-MU))))

(defthm INST-stg-step-INST-MU-RS1
    (implies (equal (INST-stg i) '(MU RS1))
             (equal (INST-stg (step-INST i MT MA sigs))
                    (if (b1p (issue-MU-RS1? (MA-MU MA) MA))
                        '(MU lch1) '(MU RS1))))
  :hints (("goal" :in-theory (enable step-INST-opener  step-INST-execute
                                     step-INST-MU))))

(defthm INST-stg-step-INST-MU-lch1
    (implies (equal (INST-stg i) '(MU lch1))
     (equal (INST-stg (step-INST i MT MA sigs))
            (if (or (b1p (CDB-FOR-MU? MA))
                    (b1p (B-NOT (MU-LATCH2-VALID? (MU-LCH2 (MA-MU MA))))))
                '(MU lch2) '(MU lch1))))
  :hints (("goal" :in-theory (enable step-INST-opener  step-INST-execute
                                     lift-b-ops step-INST-MU))))

(defthm INST-stg-step-INST-MU-lch2
   (implies (equal (INST-stg i) '(MU lch2))
     (equal (INST-stg (step-INST i MT MA sigs))
            (if (b1p (CDB-FOR-MU? MA))
                '(complete) '(MU lch2))))
  :hints (("goal" :in-theory (enable step-INST-opener  step-INST-execute
                                     lift-b-ops step-INST-MU))))

(defthm INST-stg-step-INST-BU-RS0
    (implies (equal (INST-stg i) '(BU RS0))
             (equal (INST-stg (step-INST i MT MA sigs))
                    (if (b1p (issue-BU-RS0? (MA-BU MA) MA))
                        '(complete) '(BU RS0))))
  :hints (("goal" :in-theory (enable step-INST-opener  step-INST-execute
                                     step-INST-BU))))

(defthm INST-stg-step-INST-BU-RS1
    (implies (equal (INST-stg i) '(BU RS1))
             (equal (INST-stg (step-INST i MT MA sigs))
                    (if (b1p (issue-BU-RS1? (MA-BU MA) MA))
                        '(complete) '(BU RS1))))
  :hints (("goal" :in-theory (enable step-INST-opener  step-INST-execute
                                     step-INST-BU))))

(defthm INST-stg-step-INST-LSU-RS0
    (implies (equal (INST-stg i) '(LSU RS0))
             (equal (INST-stg (step-INST i MT MA sigs))
                    (if (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                        (if (b1p (LSU-RS-ld-st? (LSU-RS0 (MA-LSU MA))))
                            (if (b1p (INST-select-wbuf0? MA sigs))
                                '(LSU wbuf0)
                                '(LSU wbuf1))
                            '(LSU rbuf))
                        '(LSU RS0))))
  :hints (("goal" :in-theory (enable step-INST-opener  step-INST-execute
                                     step-INST-LSU
                                     step-INST-LSU-RS0))))
```

```
(defthm INST-stg-step-INST-LSU-RS1
    (implies (equal (INST-stg i) '(LSU RS1))
             (equal (INST-stg (step-INST i MT MA sigs))
                    (if (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                        (if (b1p (LSU-RS-ld-st? (LSU-RS1 (MA-LSU MA))))
                            (if (b1p (INST-select-wbuf0? MA sigs))
                                '(LSU wbuf0)
                                '(LSU wbuf1))
                            '(LSU rbuf))
                        '(LSU RS1))))
  :hints (("goal" :in-theory (enable step-INST-opener  step-INST-execute
                                     step-INST-LSU step-INST-LSU-RS1))))

(defthm INST-stg-step-INST-LSU-wbuf1
    (implies (equal (INST-stg i) '(LSU wbuf1))
             (equal (INST-stg (step-INST i MT MA sigs))
                    (if (b1p (check-wbuf1? (MA-LSU MA)))
                        '(LSU wbuf1 lch)
                        (if (b1p (release-wbuf0? (MA-LSU MA) sigs))
                            '(LSU wbuf0)
                            '(LSU wbuf1)))))
  :hints (("goal" :in-theory (enable step-INST-opener  step-INST-execute
                                     step-INST-LSU step-INST-LSU-wbuf1))))

(defthm INST-stg-step-INST-LSU-wbuf0
    (implies (equal (INST-stg i) '(LSU wbuf0))
             (equal (INST-stg (step-INST i MT MA sigs))
                    (if (b1p (check-wbuf0? (MA-LSU MA)))
                        '(LSU wbuf0 lch)
                        '(LSU wbuf0))))
  :hints (("goal" :in-theory (enable step-INST-opener  step-INST-execute
                                     step-INST-LSU step-INST-LSU-wbuf0))))

(defthm INST-stg-step-INST-LSU-rbuf
    (implies (equal (INST-stg i) '(LSU rbuf))
             (equal (INST-stg (step-INST i MT MA sigs))
                    (if (b1p (release-rbuf? (MA-LSU MA) MA sigs))
                        '(LSU  lch)
                        '(LSU rbuf))))
  :hints (("goal" :in-theory (enable step-INST-opener  step-INST-execute
                                     lift-b-ops
                                     step-INST-LSU step-INST-LSU-rbuf))))

(defthm INST-stg-step-INST-LSU-wbuf0-lch
    (implies (equal (INST-stg i) '(LSU wbuf0 lch))
             (equal (INST-stg (step-INST i MT MA sigs))
                    '(complete wbuf0)))
  :hints (("goal" :in-theory (enable step-INST-opener  step-INST-execute
                                     step-INST-LSU step-INST-LSU-wbuf0-lch
                                     lift-b-ops))))

(defthm INST-stg-step-INST-LSU-wbuf1-lch
    (implies (equal (INST-stg i) '(LSU wbuf1 lch))
             (equal (INST-stg (step-INST i MT MA sigs))
                    (if (b1p (release-wbuf0? (MA-LSU MA) sigs))
                        '(complete wbuf0)
                        '(complete wbuf1))))
  :hints (("goal" :in-theory (enable step-INST-opener  step-INST-execute
                                     step-INST-LSU step-INST-LSU-wbuf1-lch
                                     lift-b-ops))))

(defthm INST-stg-step-INST-complete-normal
```

```
    (implies (equal (INST-stg i) '(complete))
             (equal (INST-stg (step-INST i MT MA sigs))
                    (if (b1p (INST-commit? i MA)) '(retire) '(complete))))
  :hints (("goal" :in-theory (enable step-INST-opener  step-INST-complete
                                     lift-b-ops))))

(defthm INST-stg-step-INST-complete-wbuf0
    (implies (equal (INST-stg i) '(complete wbuf0))
             (equal (INST-stg (step-INST i MT MA sigs))
                    (if (b1p (b-and (INST-commit? i MA) (enter-excpt? MA)))
                        '(retire)
                        (if (b1p (INST-commit? i MA))
                            '(commit wbuf0)
                            '(complete wbuf0)))))
  :hints (("goal" :in-theory (enable step-INST-opener  step-INST-complete
                                     lift-b-ops))))

(defthm INST-stg-step-INST-complete-wbuf1
    (implies (equal (INST-stg i) '(complete wbuf1))
             (equal (INST-stg (step-INST i MT MA sigs))
                    (if (b1p (b-and (INST-commit? i MA) (enter-excpt? MA)))
                        '(retire)
                        (if (b1p (b-and (INST-commit? i MA)
                                        (release-wbuf0? (MA-LSU MA) sigs)))
                            '(commit wbuf0)
                            (if (b1p (INST-commit? i MA))
                                '(commit wbuf1)
                                (if (b1p (release-wbuf0? (MA-LSU MA) sigs))
                                    '(complete wbuf0)
                                    '(complete wbuf1)))))))
  :hints (("goal" :in-theory (enable step-INST-opener  step-INST-complete
                                     lift-b-ops))))

(defthm INST-stg-step-INST-commit-wbuf0
    (implies (equal (INST-stg i) '(commit wbuf0))
             (equal (INST-stg (step-INST i MT MA sigs))
                    (if (b1p (release-wbuf0? (MA-LSU MA) sigs))
                        '(retire)
                        '(commit wbuf0))))
  :hints (("goal" :in-theory (enable step-INST-opener  step-INST-commit
                                     step-INST-LSU
                                     lift-b-ops))))

(defthm INST-stg-step-INST-commit-wbuf1
    (implies (equal (INST-stg i) '(commit wbuf1))
             (equal (INST-stg (step-INST i MT MA sigs))
                    (if (b1p (release-wbuf0? (MA-LSU MA) sigs))
                        '(commit wbuf0)
                        '(commit wbuf1))))
  :hints (("goal" :in-theory (enable step-INST-opener  step-INST-commit
                                     step-INST-LSU
                                     lift-b-ops))))

(deflabel end-inst-stg-step-inst)
(deftheory inst-stg-step-inst
    (set-difference-theories (universal-theory 'end-inst-stg-step-inst)
                             (universal-theory 'begin-inst-stg-step-inst)))

(in-theory (disable inst-stg-step-inst))

;;;;; Lemmas about Reachable Stages
; instruction can i is in stage (IU RS0) only if its previous stage is
```

```
; (DQ 0) or (IU RS0) itself.
;
; Similar lemmas follow.
(defthm stages-reachable-to-IU-RS0
    (implies (equal (INST-stg (step-INST i MT MA sigs)) '(IU RS0))
             (or (equal (INST-stg i) '(DQ 0))
                 (equal (INST-stg i) '(IU RS0))))
  :hints (("goal" :in-theory (enable step-INST MT-def-non-rec-functions
                                     DQ-stg-p)))
  :rule-classes nil)

(defthm stages-reachable-to-IU-RS1
    (implies (equal (INST-stg (step-INST i MT MA sigs)) '(IU RS1))
             (or (equal (INST-stg i) '(DQ 0))
                 (equal (INST-stg i) '(IU RS1))))
  :hints (("goal" :in-theory (enable step-INST MT-def-non-rec-functions
                                     DQ-stg-p)))
  :rule-classes nil)

(defthm stages-reachable-to-MU-RS0
    (implies (equal (INST-stg (step-INST i MT MA sigs)) '(MU RS0))
             (or (equal (INST-stg i) '(DQ 0))
                 (equal (INST-stg i) '(MU RS0))))
  :hints (("goal" :in-theory (enable step-INST MT-def-non-rec-functions
                                     DQ-stg-p)))
  :rule-classes nil)

(defthm stages-reachable-to-MU-RS1
    (implies (equal (INST-stg (step-INST i MT MA sigs)) '(MU RS1))
             (or (equal (INST-stg i) '(DQ 0))
                 (equal (INST-stg i) '(MU RS1))))
  :hints (("goal" :in-theory (enable step-INST MT-def-non-rec-functions
                                     DQ-stg-p)))
  :rule-classes nil)

(defthm stages-reachable-to-MU-lch1
    (implies (equal (INST-stg (step-INST i MT MA sigs)) '(MU lch1))
             (or (equal (INST-stg i) '(MU RS0))
                 (equal (INST-stg i) '(MU RS1))
                 (equal (INST-stg i) '(MU lch1))))
  :hints (("goal" :in-theory (enable step-INST MT-def-non-rec-functions
                                     MU-stg-p)))
  :rule-classes nil)

(defthm stages-reachable-to-MU-lch2
    (implies (equal (INST-stg (step-INST i MT MA sigs)) '(MU lch2))
             (or (equal (INST-stg i) '(MU lch1))
                 (equal (INST-stg i) '(MU lch2))))
  :hints (("goal" :in-theory (enable step-INST MT-def-non-rec-functions
                                     MU-stg-p)))
  :rule-classes nil)

(defthm stages-reachable-to-BU-RS0
    (implies (equal (INST-stg (step-INST i MT MA sigs)) '(BU RS0))
             (or (equal (INST-stg i) '(DQ 0))
                 (equal (INST-stg i) '(BU RS0))))
  :hints (("goal" :in-theory (enable step-INST MT-def-non-rec-functions
                                     DQ-stg-p BU-stg-p)))
  :rule-classes nil)

(defthm stages-reachable-to-BU-RS1
    (implies (equal (INST-stg (step-INST i MT MA sigs)) '(BU RS1))
```

```
            (or (equal (INST-stg i) '(DQ 0))
                (equal (INST-stg i) '(BU RS1))))
  :hints (("goal" :in-theory (enable step-INST MT-def-non-rec-functions
                                     DQ-stg-p BU-stg-p)))
  :rule-classes nil)

(defthm stages-reachable-to-lsu-RS0
    (implies (equal (INST-stg (step-INST i MT MA sigs)) '(LSU RS0))
             (or (equal (INST-stg i) '(DQ 0))
                 (equal (INST-stg i) '(LSU RS0))))
  :hints (("goal" :in-theory (enable step-INST MT-def-non-rec-functions
                                     DQ-stg-p LSU-stg-p)))
  :rule-classes nil)

(defthm stages-reachable-to-lsu-RS1
    (implies (equal (INST-stg (step-INST i MT MA sigs)) '(LSU RS1))
             (or (equal (INST-stg i) '(DQ 0))
                 (equal (INST-stg i) '(LSU RS1))))
  :hints (("goal" :in-theory (enable step-INST MT-def-non-rec-functions
                                     DQ-stg-p LSU-stg-p)))
  :rule-classes nil)

(defthm stages-reachable-to-lsu-wbuf1
    (implies (equal (INST-stg (step-INST i MT MA sigs)) '(LSU wbuf1))
             (or (equal (INST-stg i) '(LSU RS0))
                 (equal (INST-stg i) '(LSU RS1))
                 (equal (INST-stg i) '(LSU wbuf1))))
  :hints (("goal" :in-theory (enable step-INST MT-def-non-rec-functions
                                     LSU-stg-p)))
  :rule-classes nil)

(defthm stages-reachable-to-lsu-wbuf0
    (implies (equal (INST-stg (step-INST i MT MA sigs)) '(LSU wbuf0))
             (or (equal (INST-stg i) '(LSU RS0))
                 (equal (INST-stg i) '(LSU RS1))
                 (equal (INST-stg i) '(LSU wbuf0))
                 (equal (INST-stg i) '(LSU wbuf1))))
  :hints (("goal" :in-theory (enable step-INST MT-def-non-rec-functions
                                     LSU-stg-p)))
  :rule-classes nil)

(defthm stages-reachable-to-lsu-rbuf
    (implies (equal (INST-stg (step-INST i MT MA sigs)) '(LSU rbuf))
             (or (equal (INST-stg i) '(LSU RS0))
                 (equal (INST-stg i) '(LSU RS1))
                 (equal (INST-stg i) '(LSU rbuf))))
  :hints (("goal" :in-theory (enable step-INST MT-def-non-rec-functions
                                     LSU-stg-p)))
  :rule-classes nil)

(defthm stages-reachable-to-lsu-wbuf0-lch
    (implies (equal (INST-stg (step-INST i MT MA sigs))
                    '(LSU wbuf0 lch))
             (equal (INST-stg i) '(LSU wbuf0)))
  :hints (("goal" :in-theory (enable step-INST MT-def-non-rec-functions
                                     LSU-stg-p)))
  :rule-classes nil)

(defthm stages-reachable-to-lsu-wbuf1-lch
    (implies (equal (INST-stg (step-INST i MT MA sigs))
                    '(LSU wbuf1 lch))
             (equal (INST-stg i) '(LSU wbuf1)))
```

```
  :hints (("goal" :in-theory (enable step-INST MT-def-non-rec-functions
                                     LSU-stg-p)))
  :rule-classes nil)

(defthm stages-reachable-to-lsu-lch
    (implies (equal (INST-stg (step-INST i MT MA sigs))
                    '(LSU lch))
             (equal (INST-stg i) '(LSU rbuf)))
  :hints (("goal" :in-theory (enable step-INST MT-def-non-rec-functions
                                     LSU-stg-p)))
  :rule-classes nil)

(defthm reachable-stages-to-LSU-issued-stg-p
    (implies (LSU-issued-stg-p (INST-stg (step-INST i MT MA sigs)))
             (or (LSU-stg-p (INST-stg i))
                 (LSU-issued-stg-p (INST-stg i))))
  :hints (("Goal" :in-theory (enable LSU-stg-p LSU-issued-stg-p
                                     COMMIT-STG-P lift-b-ops
                                     COMPLETE-STG-P
                                     execute-stg-p
                                     step-INST MT-def-non-rec-functions)))
  :rule-classes nil)

; An instruction can be in an execute stage if its previous stage is
; (DQ 0) or one of the execute stages.
(defthm stages-reachable-to-execute
    (implies (and (INST-p i)
                  (execute-stg-p (INST-stg (step-INST i MT MA sigs))))
             (or (equal (INST-stg i) '(DQ 0))
                 (execute-stg-p (INST-stg i))))
  :hints (("goal" :in-theory (enable step-INST MT-def-non-rec-functions
                                     DQ-stg-p)))
  :rule-classes nil)

; An instruction can be in a complete stage if its previous stage is
; (DQ 0) or one of the execute stages.
(defthm stages-reachable-to-complete
    (implies (and (INST-p i)
                  (complete-stg-p (INST-stg (step-INST i MT MA sigs))))
             (or (equal (INST-stg i) '(DQ 0))
                 (execute-stg-p (INST-stg i))
                 (complete-stg-p (INST-stg i))))
  :hints (("goal" :in-theory (enable step-INST MT-def-non-rec-functions
                                     DQ-stg-p)))
  :rule-classes nil)

(defthm stages-reachable-to-complete-normal
    (implies (and (INST-p i)
                  (equal (INST-stg (step-INST i MT MA sigs)) '(complete)))
             (or (equal (INST-stg i) '(DQ 0))
                 (equal (INST-stg i) '(IU RS0))
                 (equal (INST-stg i) '(IU RS1))
                 (equal (INST-stg i) '(MU lch2))
                 (equal (INST-stg i) '(BU RS0))
                 (equal (INST-stg i) '(BU RS1))
                 (equal (INST-stg i) '(LSU lch))
                 (equal (INST-stg i) '(complete))))
  :hints (("goal" :in-theory (enable step-INST DQ-stg-p
                                     execute-stg-p IU-stg-p MU-stg-p
                                     BU-stg-p
                                     MT-def-non-rec-functions)))
  :rule-classes nil)
```

```
(defthm stages-reachable-to-complete-wbuf0
    (implies (and (INST-p i)
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(complete wbuf0)))
             (or (equal (INST-stg i) '(LSU wbuf0 lch))
                 (equal (INST-stg i) '(LSU wbuf1 lch))
                 (equal (INST-stg i) '(complete wbuf0))
                 (equal (INST-stg i) '(complete wbuf1))))
  :hints (("goal" :in-theory (enable step-INST
                                     EXECUTE-STG-P LSU-stg-p
                                     complete-stg-p
                                     MT-def-non-rec-functions)))
  :rule-classes nil)

(defthm stages-reachable-to-complete-wbuf1
    (implies (and (INST-p i)
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(complete wbuf1)))
             (or (equal (INST-stg i) '(LSU wbuf1 lch))
                 (equal (INST-stg i) '(complete wbuf1))))
  :hints (("goal" :in-theory (enable step-INST
                                     execute-stg-p LSU-stg-p
                                     complete-stg-p
                                     MT-def-non-rec-functions)))
  :rule-classes nil)

(defthm stages-reachable-to-commit-wbuf0
    (implies (and (INST-p i)
                  (equal (INST-stg (step-INST i MT MA sigs)) '(commit wbuf0)))
             (or (equal (INST-stg i) '(complete wbuf0))
                 (equal (INST-stg i) '(complete wbuf1))
                 (equal (INST-stg i) '(commit wbuf0))
                 (equal (INST-stg i) '(commit wbuf1))))
  :hints (("goal" :in-theory (enable step-INST
                                     complete-stg-p
                                     commit-stg-p
                                     MT-def-non-rec-functions)))
  :rule-classes nil)

(defthm stages-reachable-to-commit-wbuf1
    (implies (and (INST-p i)
                  (equal (INST-stg (step-INST i MT MA sigs)) '(commit wbuf1)))
             (or (equal (INST-stg i) '(complete wbuf1))
                 (equal (INST-stg i) '(commit wbuf1))))
  :hints (("goal" :in-theory (enable step-INST
                                     complete-stg-p
                                     MT-def-non-rec-functions)))
  :rule-classes nil)

(defthm stages-reachable-to-retire-stg
    (implies (retire-stg-p (INST-stg (step-INST i MT MA sigs)))
             (or (equal (INST-stg i) '(retire))
                 (equal (INST-stg i) '(complete))
                 (equal (INST-stg i) '(complete wbuf0))
                 (equal (INST-stg i) '(complete wbuf1))
                 (equal (INST-stg i) '(commit wbuf0))
                 (equal (INST-stg i) '(commit wbuf1))))
  :hints (("Goal" :in-theory (enable step-INST MT-def-non-rec-functions
                                     complete-stg-p
                                     RETIRE-STG-P)))
  :rule-classes nil)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; INST-functions and step-INST
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defthm INST-ID-fetched-inst
    (equal (INST-ID (fetched-inst MT ISA spc smc)) (MT-new-ID MT))
  :hints (("Goal" :in-theory (enable MT-def))))

(defthm INST-ID-exintr-INST
    (equal (INST-ID (exintr-INST MT ISA spc)) (MT-new-ID MT))
  :hints (("Goal" :in-theory (enable MT-def))))

(defthm INST-ID-step-INST
    (equal (INST-ID (step-INST i MT MA sigs)) (INST-ID i))
  :hints (("Goal" :in-theory (enable MT-def))))



(defthm INST-modified-exintr-INST
    (equal (INST-modified? (exintr-INST MT pre smc)) smc)
  :hints (("Goal" :in-theory (enable exintr-INST))))

(defthm INST-modified-fetched-inst
    (equal (INST-modified? (fetched-inst MT ISA spc smc))
           (if (MT-no-write-at (ISA-pc ISA) MT)
               (bfix smc) 1))
  :hints (("Goal" :in-theory (enable fetched-inst lift-b-ops))))

(defthm INST-modified-step-INST
    (implies (INST-p i)
             (equal (INST-modified? (step-INST i MT MA sigs))
                    (INST-modified? i)))
  :hints (("Goal" :in-theory (enable MT-def))))

(defthm INST-first-modified-step-INST
    (implies (INST-p i)
             (equal (INST-first-modified? (step-INST i MT MA sigs))
                    (INST-first-modified? i)))
  :hints (("Goal" :in-theory (enable MT-def))))

(defthm inst-first-modified-fetched-inst
    (equal (INST-first-modified? (fetched-inst MT ISA spc smc))
           (b-nor smc (if (MT-no-write-at (ISA-pc ISA) MT) 1 0)))
  :hints (("Goal" :in-theory (enable fetched-inst equal-b1p-converter
                                     lift-b-ops))))

(defthm inst-specultv-fetched-inst
    (equal (inst-specultv? (fetched-inst MT ISA spc smc))
           spc)
  :hints (("Goal" :in-theory (enable MT-def))))


(defthm inst-specultv-exintr-INST
    (equal (inst-specultv? (exintr-INST MT ISA smc)) 0)
  :hints (("Goal" :in-theory (enable MT-def))))

(defthm inst-specultv-step-INST
    (equal (inst-specultv? (step-INST i MT MA sigs))
           (inst-specultv? i))
  :hints (("Goal" :in-theory (enable MT-def))))

(defthm INST-br-predict-fetched-inst
    (equal (INST-br-predict? (fetched-inst MT ISA flg flg2)) 0)
```

```
  :hints (("Goal" :in-theory (enable fetched-inst))))

(defthm INST-br-predict-exintr-INST
    (equal (INST-br-predict? (exintr-INST MT ISA flg)) 0)
  :hints (("Goal" :in-theory (enable exintr-INST))))

; Field br-predict? of a control vector stores the branch prediction outcome.
(defthm cntlv-br-predict-INST-cntlv
    (implies (INST-p i)
             (equal (cntlv-br-predict? (INST-cntlv i))
                    (INST-br-predict? i)))
  :hints (("goal" :in-theory (enable INST-cntlv))))

(defthm cntlv-sync-and-branch-of-INST-cntlv-exclusive
    (implies (and (INST-p i)
                  (b1p (cntlv-sync? (INST-cntlv i))))
             (equal (logbit 3 (cntlv-exunit (INST-cntlv i))) 0))
  :hints (("goal" :in-theory (enable INST-cntlv))))

; INST-exintr
(defthm INST-exintr-fetched-inst
    (equal (INST-exintr? (fetched-inst MT ISA flg flg2)) 0)
  :hints (("Goal" :in-theory (enable fetched-inst))))

(defthm INST-exintr-step-INST
    (equal (INST-exintr? (step-INST i MT MA sigs)) (INST-exintr? i))
  :hints (("Goal" :in-theory (enable step-INST MT-def))))

(defthm INST-exintr-exintr-INST
    (equal (INST-exintr? (exintr-INST MT ISA smc)) 1)
  :hints (("Goal" :in-theory (enable exintr-INST))))

(defthm INST-word-fetched-inst
    (equal (INST-word (fetched-inst MT ISA spc smc))
           (read-mem (ISA-pc ISA) (ISA-mem ISA)))
  :Hints (("goal" :in-theory (enable fetched-inst INST-word
                                     INST-pc INST-mem))))

(defthm Inst-word-step-INST
    (equal (Inst-word (step-INST i MT MA sigs)) (Inst-word i))
  :hints (("Goal" :in-theory (enable MT-def ))))

(defthm Inst-opcode-step-INST
    (equal (Inst-opcode (step-INST i MT MA sigs)) (Inst-opcode i))
  :hints (("Goal" :in-theory (enable MT-def ))))

(defthm Inst-ra-step-INST
    (equal (Inst-ra (step-INST i MT MA sigs)) (Inst-ra i))
  :hints (("Goal" :in-theory (enable MT-def ))))

(defthm Inst-rb-step-INST
    (equal (Inst-rb (step-INST i MT MA sigs)) (Inst-rb i))
  :hints (("Goal" :in-theory (enable MT-def ))))

(defthm Inst-rc-step-INST
    (equal (Inst-rc (step-INST i MT MA sigs)) (Inst-rc i))
  :hints (("Goal" :in-theory (enable MT-def ))))

(defthm Inst-im-step-INST
    (equal (Inst-im (step-INST i MT MA sigs)) (Inst-im i))
  :hints (("Goal" :in-theory (enable MT-def ))))
```

```
(defthm INST-br-predict-step-non-IFU-INST
    (implies (not (IFU-stg-p (INST-stg i)))
             (equal (INST-br-predict? (step-INST i MT MA sigs))
                    (INST-br-predict? i)))
  :hints (("goal" :in-theory (enable MT-def))))

(defthm INST-br-predict-step-INST
    (implies (and (not (b1p (DQ-full? (MA-DQ MA))))
                  (IFU-stg-p (INST-stg i)))
             (equal (INST-br-predict? (step-INST i MT MA sigs))
                    (IFU-branch-predict? (MA-IFU MA) MA sigs)))
  :hints (("goal" :in-theory (enable step-INST step-INST-IFU
                                     equal-b1p-converter))))


(defthm INST-writeback-p-iff-INST-wb
    (iff (INST-writeback-p i) (b1p (INST-wb? i)))
  :rule-classes
  ((:rewrite :corollary
             (implies (b1p (INST-wb? i)) (INST-writeback-p i)))
   (:rewrite :corollary
             (implies (not (b1p (INST-wb? i))) (not (INST-writeback-p i)))))
  :hints (("goal" :in-theory (enable INST-writeback-p INST-wb? lift-b-ops
                                     INST-opcode INST-cntlv
                                     decode rdb logbit*))))

; Following lemmas show that return values of INST functions don't change
; after step-INST.
;
; For example, (INST-writeback-p i') = (INST-writeback i) if i'
; is INST i updated by step-INST.
(defthm INST-writeback-p-step-INST
    (implies (INST-p i)
             (equal (INST-writeback-p (step-INST i MT MA sigs))
                    (INST-writeback-p i)))
  :hints (("goal" :in-theory (enable INST-writeback-p))))

(defthm INST-cntlv-step-inst-if-IFU-stg
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (INST-p i)
                  (IFU-stg-p (INST-stg i))
                  (not (b1p (DQ-full? (MA-DQ MA)))))
             (equal (INST-cntlv (step-INST i MT MA sigs))
                    (decode (INST-opcode i)
                            (IFU-branch-predict? (MA-IFU MA) MA sigs))))
  :hints (("goal" :in-theory (enable  inst-cntlv))))

(defthm INST-cntlv-step-non-IFU-INST
    (implies (not (IFU-stg-p (INST-stg i)))
             (equal (INST-cntlv (step-INST i MT MA sigs))
                    (INST-cntlv i)))
  :hints (("goal" :in-theory (enable INST-cntlv))))

(defthm INST-IU-step-inst
    (implies (and (INST-p i) (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (equal (INST-IU? (step-INST i MT MA sigs))
                    (INST-IU? i)))
  :hints (("goal" :in-theory (enable INST-function-def
                                     INST-cntlv decode lift-b-ops
                                     rdb logbit*))))
```

```
(defthm INST-no-unit-step-inst
    (implies (and (INST-p i) (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (equal (INST-no-unit? (step-INST i MT MA sigs))
                    (INST-no-unit? i)))
  :hints (("goal" :in-theory (enable INST-function-def
                                     INST-cntlv decode lift-b-ops
                                     rdb logbit*))))

(defthm INST-BU-step-inst
    (implies (and (INST-p i) (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (equal (INST-BU? (step-INST i MT MA sigs))
                    (INST-BU? i)))
  :hints (("goal" :in-theory (enable INST-function-def
                                     INST-cntlv decode lift-b-ops
                                     rdb logbit*))))

(defthm INST-LSU-step-inst
    (implies (and (INST-p i) (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (equal (INST-LSU? (step-INST i MT MA sigs))
                    (INST-LSU? i)))
  :hints (("goal" :in-theory (enable INST-function-def
                                     INST-cntlv decode lift-b-ops
                                     rdb logbit*))))

(defthm INST-MU-step-inst
    (implies (and (INST-p i) (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (equal (INST-MU? (step-INST i MT MA sigs))
                    (INST-MU? i)))
  :hints (("goal" :in-theory (enable INST-function-def
                                     INST-cntlv decode lift-b-ops
                                     rdb logbit*))))

(defthm INST-ld-st-step-inst
    (equal (INST-ld-st? (step-INST i MT MA sigs))
           (INST-ld-st? i))
  :hints (("goal" :in-theory (enable INST-function-def
                                     INST-cntlv decode lift-b-ops
                                     rdb logbit*))))

(defthm INST-sync-step-inst
    (equal (INST-sync? (step-INST i MT MA sigs))
           (INST-sync? i))
  :hints (("goal" :in-theory (enable INST-sync? INST-cntlv decode lift-b-ops
                                     rdb logbit*))))

(defthm INST-wb-step-inst
    (equal (INST-wb? (step-INST i MT MA sigs))
           (INST-wb? i))
  :hints (("goal" :in-theory (enable INST-function-def
                                     INST-cntlv decode lift-b-ops
                                     rdb logbit*))))

(defthm INST-wb-sreg-step-inst
    (equal (INST-wb-sreg? (step-INST i MT MA sigs))
           (INST-wb-sreg? i))
  :hints (("goal" :in-theory (enable INST-function-def
                                     INST-cntlv decode lift-b-ops
                                     rdb logbit*))))

(defthm INST-rfeh-step-inst
    (equal (INST-rfeh? (step-INST i MT MA sigs))
```

```
          (INST-rfeh? i))
  :hints (("goal" :in-theory (enable INST-function-def
                                     INST-cntlv decode lift-b-ops
                                     rdb logbit*))))

(defthm INST-dest-reg-step-INST
    (equal (Inst-dest-reg (step-INST i MT MA sigs)) (Inst-dest-reg i))
  :hints (("Goal" :in-theory (enable MT-def ))))

(defthm INST-tag-step-INST
    (implies (and (INST-p i) (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (not (DQ-stg-p (INST-stg i))))
             (equal (INST-tag (step-INST i MT MA sigs)) (INST-tag i)))
  :hints (("goal" :in-theory (enable step-INST
                                     step-INST-low-level-functions))))

(defthm INST-pre-ISA-step-INST
    (equal (INST-pre-ISA (step-INST i MT MA sigs))
           (INST-pre-ISA i))
  :hints (("goal" :in-theory (enable MT-def))))

(defthm INST-post-ISA-step-INST
    (equal (INST-post-ISA (step-INST i MT MA sigs))
           (INST-post-ISA i))
  :hints (("goal" :in-theory (enable MT-def))))

(defthm INST-pre-ISA-fetched-inst
    (equal (INST-pre-ISA (fetched-inst MT ISA spc smc)) ISA)
  :hints (("goal" :in-theory (enable MT-def))))

(defthm INST-post-ISA-fetched-inst
    (equal (INST-post-ISA (fetched-inst MT ISA spc smc))
           (ISA-step ISA (ISA-input 0)))
  :hints (("goal" :in-theory (enable MT-def))))

(defthm INST-pre-ISA-exintr-INST
    (equal (INST-pre-ISA (exintr-INST MT ISA smc)) ISA)
  :hints (("goal" :in-theory (enable MT-def))))

(defthm INST-post-ISA-exintr-INST
    (equal (INST-post-ISA (exintr-INST MT ISA smc))
           (ISA-step ISA (ISA-input 1)))
  :hints (("goal" :in-theory (enable MT-def))))

(defthm INST-pc-step-INST
    (equal (INST-pc (step-INST i MT MA sigs)) (INST-pc i))
  :hints (("Goal" :in-theory (enable MT-def))))

(defthm INST-su-step-INST
    (equal (INST-su (step-INST i MT MA sigs)) (INST-su i))
  :hints (("Goal" :in-theory (enable MT-def))))

(defthm INST-br-target-step
    (equal (INST-br-target (step-INST i MT MA sigs)) (INST-br-target i))
  :hints (("goal" :in-theory (enable INST-br-target))))

(defthm INST-branch-taken-step-inst
    (equal (INST-branch-taken? (step-INST i MT MA sigs))
           (INST-branch-taken? i))
  :hints (("goal" :in-theory (enable INST-branch-taken?))))
```

```
(defthm INST-src-val1-step-INST
    (equal (INST-src-val1 (step-INST i MT MA sigs)) (INST-src-val1 i))
  :hints (("Goal" :in-theory (enable MT-def))))

(defthm INST-src-val2-step-INST
    (equal (INST-src-val2 (step-INST i MT MA sigs)) (INST-src-val2 i))
  :hints (("Goal" :in-theory (enable MT-def))))

(defthm INST-src-val3-step-INST
    (equal (INST-src-val3 (step-INST i MT MA sigs)) (INST-src-val3 i))
  :hints (("Goal" :in-theory (enable MT-def))))

(defthm INST-dest-val-step-INST
    (equal (INST-dest-val (step-INST i MT MA sigs))
           (INST-dest-val i))
  :hints (("goal" :in-theory (enable INST-function-def))))

(defthm INST-IU-op-step-INST
    (implies (and (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs))
             (equal (INST-IU-op? (step-INST i MT MA sigs)) (INST-IU-op? i)))
  :hints (("goal" :in-theory (enable  inst-function-def inst-cntlv
                                      decode lift-b-ops rdb logbit*))))

(defthm INST-LSU-op-step-INST
    (implies (and (INST-p i) (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (equal (INST-LSU-op? (step-INST i MT MA sigs)) (INST-LSU-op? i)))
  :hints (("goal" :in-theory (enable  inst-function-def inst-cntlv
                                      decode lift-b-ops rdb logbit*))))

(defthm INST-load-addr-step-inst
    (equal (INST-load-addr (step-INST i MT MA sigs))
           (INST-load-addr i))
  :hints (("Goal" :in-theory (enable MT-def))))

(defthm INST-load-step-INST
    (implies (and (INST-p i) (MAETT-p MT)
                  (MA-state-p MA) (MA-input-p sigs))
             (equal (INST-load? (step-INST i MT MA sigs))
                    (INST-load? i)))
  :hints (("goal" :in-theory (enable INST-function-def lift-b-ops
                                     ISA-functions decode
                                     rdb logbit*))))

(defthm INST-store-step-INST
    (implies (and (INST-p i) (MAETT-p MT)
                  (MA-state-p MA) (MA-input-p sigs))
             (equal (INST-store? (step-INST i MT MA sigs))
                    (INST-store? i)))
  :hints (("goal" :in-theory (enable INST-function-def lift-b-ops
                                     ISA-functions decode
                                     rdb logbit*))))

(defthm INST-LSU-if-INST-store
    (implies (b1p (INST-store? i))
             (equal (INST-LSU? i) 1))
  :hints (("Goal" :in-theory (enable INST-LSU? lift-b-ops
                                     equal-b1p-converter
                                     b1p-bit-rewriter
                                     INST-store?))))
```

```
(defthm INST-LSU-if-INST-load
    (implies (b1p (INST-load? i))
             (equal (INST-LSU? i) 1))
  :hints (("Goal" :in-theory (enable INST-LSU? lift-b-ops
                                     equal-b1p-converter
                                     b1p-bit-rewriter
                                     INST-load?))))

(defthm INST-LSU-IF-INST-store-2
    (implies (not (b1p (INST-LSU? i))) (not (b1p (INST-store? i))))
  :hints (("goal" :In-theory (enable INST-LSU? lift-b-ops b1p-bit-rewriter
                                     INST-store?))))

(defthm INST-LSU-IF-INST-load-2
    (implies (not (b1p (INST-LSU? i)))
             (not (b1p (INST-load? i))))
  :hints (("goal" :In-theory (enable INST-LSU? lift-b-ops b1p-bit-rewriter
                                     INST-load?))))

(defthm INST-store-addr-step-INST
    (equal (INST-store-addr (step-INST i MT MA sigs))
           (INST-store-addr i))
  :hints (("goal" :in-theory (enable INST-function-def lift-b-ops))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Lemmas about relations between instructions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Following lemmas show that distinct instructions are not equal to each
; other after step-INST.
(encapsulate nil
(local
(defthm INST-new-ID-distinct-if-INST-in-help
    (implies (and (ID-LT-ALL-P trace new-ID)
                  (member-equal i trace))
             (< (INST-ID i) new-ID))))

(defthm INST-new-ID-distinct-if-INST-in
    (implies (and (weak-inv MT)
                  (INST-in i MT))
             (< (INST-ID i) (MT-new-ID MT)))
  :hints (("goal" :in-theory (enable weak-inv INST-in
                                     MT-new-ID-distinct-p)
                  :restrict ((INST-new-ID-distinct-if-INST-in-help
                              ((trace (MT-trace MT))))))))
  :rule-classes :linear)
)

(encapsulate nil
(local
 (defthm INST-ID-distinct-help
     (implies (and (member-equal i2 trace)
                   (equal (INST-ID i1) (INST-ID i2)))
              (member-eq-ID i1 trace))))

(local
(defthm INST-ID-distinct-local
    (implies (and (distinct-IDs-p trace)
                  (INST-listp trace)
                  (member-equal I1 trace)
                  (member-equal I2 trace)
                  (not (equal I1 I2)))
             (not (equal (INST-ID I1) (INST-ID I2))))))
```

```
(defthm INST-ID-distinct
    (implies (and (weak-inv MT)
                  (MAETT-p MT)
                  (INST-in I1 MT) (INST-in I2 MT)
                  (not (equal I1 I2)))
             (not (equal (INST-ID I1) (INST-ID I2))))
  :hints (("goal" :in-theory (enable weak-inv MT-distinct-IDs-p
                                     MAETT-p INST-in))))
)

(defthm equal-step-INSTs
    (implies (and (weak-inv MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in I1 MT)
                  (INST-in I2 MT))
             (equal (equal (step-INST I1 MT MA sigs)
                           (step-INST I2 MT MA sigs))
                    (equal I1 I2)))
  :hints (("goal" :cases ((equal (INST-ID (step-INST I1 MT MA sigs))
                                 (INST-ID (step-INST I2 MT MA sigs)))))
          ("subgoal 2" :in-theory (disable INST-ID-step-INST))))

(defthm equal-step-INST-fetched-inst
    (implies (and (weak-inv MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT))
             (not (equal (step-INST i MT MA sigs)
                         (fetched-inst MT ISA spc smc))))
  :hints (("goal" :cases ((equal (INST-ID (step-INST i MT MA sigs))
                                 (INST-ID (fetched-inst MT ISA spc smc)))))
          ("Subgoal 2" :in-theory (disable INST-ID-step-INST))))

(defthm equal-step-Inst-word-for-exintr
    (implies (and (weak-inv MT)
                  (MAETT-p MT)
                  (INST-in i MT))
             (not (equal (step-INST i MT MA sigs)
                         (exintr-INST MT ISA spc))))
  :hints (("goal" :cases ((equal (INST-ID (step-INST i MT MA sigs))
                                 (INST-ID (exintr-INST MT ISA spc)))))
          ("Subgoal 2" :in-theory (disable INST-ID-step-INST))))

(encapsulate nil
(local
 (defthm MT-modify-exintr-INST-help
     (not (trace-modify-p (exintr-INST MT ISA smc) trace))
   :hints (("goal" :in-theory (enable INST-MODIFY-P)))))

(defthm MT-modify-exintr-INST
    (not (MT-modify-p (exintr-INST MT ISA smc) MT2))
  :hints (("goal" :in-theory (enable MT-modify-p))))
)

(defthm MT-modified-p-car-MT-trace
    (implies (and (MAETT-p MT)
                  (consp (MT-trace MT)))
             (not (MT-modify-p (car (MT-trace MT)) MT)))
  :hints (("goal" :in-theory (enable MT-modify-p))))

; A non-speculative instruction is not a member of trace-all-specultv-p.
(defthm inst-specultv-is-not-member-equal-to-trace-all-specultv
```

```
    (implies (and (trace-all-specultv-p trace)
                  (not (b1p (inst-specultv? i))))
             (not (member-equal i trace))))

(encapsulate nil
(local
(defthm INST-in-order-p-inst-specultv-help-help
    (implies (and (member-equal i trace)
                  (trace-all-specultv-p trace)
                  (INST-listp trace))
             (equal (inst-specultv? i) 1))
  :hints (("goal" :in-theory (enable equal-b1p-converter)))))

(local
(defthm INST-in-order-p-inst-specultv-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (trace-correct-speculation-p trace)
                  (b1p (inst-specultv? i))
                  (not (b1p (inst-specultv? j)))
                  (INST-listp trace))
             (not (member-in-order i j trace)))
  :hints (("goal" :in-theory (enable member-in-order*
           inst-specultv-is-not-member-equal-to-trace-all-specultv)))))

; If instruction i precedes instruction j in program order, and
; if i is a speculatively executed instruction, so is j.
(defthm INST-in-order-p-inst-specultv
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA)
                  (b1p (inst-specultv? i))
                  (INST-in-order-p i j MT))
             (b1p (inst-specultv? j)))
  :hints (("goal" :use (:instance INST-in-order-p-inst-specultv-help
                                  (trace (MT-trace MT)))
                  :in-theory (enable inv correct-speculation-p
                                     INST-in INST-in-order-p)))
  :rule-classes
  ((:rewrite :corollary
             (implies (and (inv MT MA)
                           (b1p (inst-specultv? i))
                           (not (b1p (inst-specultv? j)))
                           (MAETT-p MT) (MA-state-p MA))
                      (not (INST-in-order-p i j MT))))))
)

(encapsulate nil
(local
(defthm INST-in-order-p-INST-modified-help-help
    (implies (and (member-equal i trace)
                  (trace-correct-modified-flgs-p trace MT flg)
                  (b1p flg)
                  (INST-listp trace))
             (equal (INST-modified? i) 1))
  :hints (("goal" :in-theory (enable equal-b1p-converter)))))

(local
(defthm INST-in-order-p-INST-modified-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (trace-correct-modified-flgs-p trace MT flg)
                  (b1p (INST-modified? i))
                  (not (b1p (INST-modified? j)))
```

```
                (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA))
           (not (member-in-order i j trace)))
  :hints (("goal" :in-theory (enable member-in-order*)))))

; If instruction i precedes instruction j, and (INST-modified? i) is
; true, then so is (INST-modified? j).  INST-modified? is defined in
; the same way as inst-specultv?, and (INST-modifier? i) is false only
; if no instruction before i is modified by self-modifying code.
(defthm INST-in-order-p-INST-modified
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA)
                  (b1p (INST-modified? i))
                  (INST-in-order-p i j MT))
             (b1p (INST-modified? j)))
  :hints (("goal" :use (:instance INST-in-order-p-INST-modified-help
                                  (trace (MT-trace MT))
                                  (flg 0))
                  :in-theory (enable inv weak-inv
                                     correct-modified-flgs-p
                                     INST-in INST-in-order-p)))
  :rule-classes
  ((:rewrite :corollary
             (implies (and (inv MT MA)
                           (b1p (INST-modified? i))
                           (not (b1p (INST-modified? j)))
                           (MAETT-p MT) (MA-state-p MA))
                      (not (INST-in-order-p i j MT))))))
)

(encapsulate nil
(local
(encapsulate nil
(local
(defthm trace-modify-p-if-INST-modify-p
    (implies (and (inv MT MA)
                  (member-equal i trace)
                  (member-equal j trace)
                  (member-in-order i j trace)
                  (subtrace-p trace MT)
                  (not (trace-modify-p j trace)))
             (not (INST-modify-p i j)))
  :hints (("goal" :in-theory (enable member-in-order*)))))

(defthm MT-modify-p-if-INST-modify-p
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-p j)
                  (INST-in i MT) (INST-in j MT)
                  (INST-in-order-p i j MT)
                  (not (MT-modify-p j MT)))
             (not (INST-modify-p i j)))
  :hints (("goal" :in-theory (enable INST-in INST-in-order-p
                                     MT-modify-p))))
))

(local
(encapsulate nil
(local
(defthm trace-INST-modified-if-MT-modify-p
    (implies (and (TRACE-CORRECT-MODIFIED-FLGS-P trace MT sticky)
                  (member-equal j trace)
                  (not (b1p (INST-modified? j))))
```

```
            (not (MT-modify-p j MT)))))

(defthm INST-modified-if-MT-modify-p
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p j)
                  (INST-in j MT)
                  (not (b1p (INST-modified? j))))
             (not (MT-modify-p j MT)))
  :hints (("goal" :in-theory (enable inv weak-inv INST-in
                                     correct-modified-flgs-p))))
))

(defthm INST-modified-if-INST-modify-p
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-p j)
                  (INST-in i MT) (INST-in j MT)
                  (INST-in-order-p i j MT)
                  (INST-modify-p i j))
             (b1p (INST-modified? j)))
  :rule-classes
  ((:rewrite :corollary
             (implies (and (inv MT MA)
                           (MAETT-p MT) (MA-state-p MA)
                           (INST-p i) (INST-p j)
                           (INST-in i MT) (INST-in j MT)
                           (INST-modify-p i j)
                           (not (b1p (INST-modified? j))))
                      (not (INST-in-order-p i j MT))))))

)
(in-theory (disable INST-modified-if-INST-modify-p))

(encapsulate nil
(local
(defthm inst-in-order-p-INST-start-specultv-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (trace-correct-speculation-p trace)
                  (b1p (INST-start-specultv? i))
                  (not (b1p (inst-specultv? j)))
                  (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA))
             (not (member-in-order i j trace)))
  :hints (("goal" :in-theory (enable member-in-order*)))
  :rule-classes nil))

; Suppose instruction i precedes instruction j.  If i is not committed,
; and if i starts a speculative execution, j is speculatively executed
; instruction.
(defthm inst-in-order-p-INST-start-specultv
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA)
                  (b1p (INST-start-specultv? i))
                  (INST-in-order-p i j MT))
             (b1p (inst-specultv? j)))
  :hints (("goal" :in-theory (enable inv correct-speculation-p
                                     INST-in-order-p)
                  :use (:instance inst-in-order-p-INST-start-specultv-help
                                  (trace (MT-trace MT))
                                  (MT MT) (MA MA))))
  :rule-classes
```

```
  ((:rewrite :corollary
             (implies (and (inv MT MA)
                           (b1p (INST-start-specultv? i))
                           (not (b1p (inst-specultv? j)))
                           (MAETT-p MT) (MA-state-p MA))
                      (not (INST-in-order-p i j MT)))))
  )
)

; Suppose trace is a subtrace of MT.  If (car trace) is a
; speculatively executed instruction, any instruction in (cdr trace)
; is also speculatively executed.
(defthm inst-specultv-car
    (implies (and (inv MT MA)
                  (member-equal i (cdr trace))
                  (consp trace)
                  (not (b1p (inst-specultv? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace))
             (equal (inst-specultv? (car trace)) 0))
  :hints (("goal" :in-theory (e/d (equal-b1p-converter)
                                  (INST-in-order-p-inst-specultv))
                  :use (:instance INST-in-order-p-inst-specultv
                                  (i (car trace)) (j i)))))

; Suppose instruction i follows instruction j=(car trace).  If j is an
; exception-raising instruction, then i is speculatively executed
; instruction.
(defthm INST-excpt-car
    (implies (and (inv MT MA)
                  (consp trace)
                  (member-equal i (cdr trace))
                  (not (b1p (inst-specultv? i)))
                  (not (committed-p (car trace)))
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace))
             (equal (INST-excpt? (car trace)) 0))
  :hints (("goal" :in-theory (e/d (equal-b1p-converter
                                   INST-start-specultv? lift-b-ops)
                                  (inst-in-order-p-INST-start-specultv))
                  :use (:instance inst-in-order-p-INST-start-specultv
                                  (i (car trace)) (j i)))))

(defthm INST-start-specultv-car
    (implies (and (inv MT MA)
                  (member-equal i (cdr trace))
                  (consp trace)
                  (not (b1p (inst-specultv? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace))
             (equal (INST-start-specultv? (car trace)) 0))
  :hints (("goal" :in-theory (e/d (equal-b1p-converter)
                                  (INST-IN-ORDER-P-INST-START-SPECULTV))
                  :use (:instance INST-IN-ORDER-P-INST-START-SPECULTV
                                  (i (car trace)) (j i)))))
(in-theory (disable INST-start-specultv-car))

; If INST-modified? is true for (car trace), INST-modified? is also true for
; any instruction in (cdr trace).
```

```
(defthm INST-modified-car
    (implies (and (inv MT MA)
                  (member-equal i (cdr trace))
                  (consp trace)
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace))
             (equal (INST-modified? (car trace)) 0))
  :hints (("goal" :in-theory (e/d (equal-b1p-converter)
                                  (INST-in-order-p-INST-modified))
                  :use (:instance INST-in-order-p-INST-modified
                                  (i (car trace))
                                  (j i)))))

; If an instruction i follows instruction j=(car trace).  If j is an
; context synchronization instruction, then i is a speculatively executed
; instruction.
(defthm INST-context-sync-car
    (implies (and (inv MT MA)
                  (consp trace)
                  (member-equal i (cdr trace))
                  (not (b1p (inst-specultv? i)))
                  (not (committed-p (car trace)))
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace))
             (equal (INST-context-sync? (car trace)) 0))
  :hints (("goal" :in-theory (e/d (equal-b1p-converter
                                   INST-start-specultv? lift-b-ops)
                                  (inst-in-order-p-INST-start-specultv))
                  :use (:instance inst-in-order-p-INST-start-specultv
                                  (i (car trace)) (j i)))))
;(in-theory (disable inst-specultv-is-not-member-equal-to-trace-all-specultv))

(defthm not-inst-modified-car-MT-trace
    (implies (and (inv MT MA)
                  (consp (MT-trace MT))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-modified? (car (MT-trace MT))) 0))
  :hints (("goal" :in-theory (enable inv weak-inv
                                     correct-modified-flgs-p)
                  :expand (TRACE-CORRECT-MODIFIED-FLGS-P (MT-TRACE MT)
                                                         MT 0))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Opening INST-inv
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deflabel begin-INST-inv-opener)
(defthm INST-inv-for-IFU
    (implies (IFU-stg-p (INST-stg i))
             (equal (INST-inv i MA)
                    (IFU-inst-inv i MA)))
  :hints (("Goal" :in-theory (enable INST-inv))))

(defthm DQ-inv-for-execute
    (implies (DQ-stg-p (INST-stg i))
             (equal (INST-inv i MA)
                    (DQ-inst-inv i MA)))
  :hints (("Goal" :in-theory (enable INST-inv))))

(defthm INST-inv-for-execute
```

```
    (implies (execute-stg-p (INST-stg i))
             (equal (INST-inv i MA)
                    (execute-inst-inv i MA)))
  :hints (("Goal" :in-theory (enable INST-inv))))

(defthm INST-inv-for-complete
    (implies (complete-stg-p (INST-stg i))
             (equal (INST-inv i MA)
                    (complete-inst-inv i MA)))
  :hints (("Goal" :in-theory (enable INST-inv))))

(defthm INST-inv-for-commit
    (implies (commit-stg-p (INST-stg i))
             (equal (INST-inv i MA)
                    (commit-inst-inv i MA)))
  :hints (("Goal" :in-theory (enable INST-inv))))
(deflabel end-INST-inv-opener)

(deftheory INST-inv-open-lemmas
    (set-difference-theories (universal-theory 'end-INST-inv-opener)
                             (universal-theory 'begin-INST-inv-opener)))

(deftheory INST-inv-opener
    (set-difference-theories
     (union-theories (theory 'inst-inv-def)
                     (theory 'INST-inv-open-lemmas))
     '(INST-inv)))

(in-theory (disable INST-inv-opener))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  MT and MA relations
;    Invariants that every instructions should satisfy
;    Relations between Exceptions
;    Exceptions and stages
;    MT-init-ISA
;    MT-DQ-len
;    Lemmas about inst-specultv?
;    Lemmas about INST-exintr?
;    Lemmas about INST-word
;    Miscellaneous lemmas about INST-pre-ISA and INST-post-ISA.
;    Relations between ISA predicate and INST predicates
;    Lemmas about subtraces and trace predicates.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(encapsulate nil
(local
(defthm INST-inv-if-member
    (implies (and (trace-INST-inv trc MA)
                  (member-equal i trc))
             (INST-inv i MA))))

(local
(defthm INST-inv-if-MT-INST-inv
    (implies (and (MT-INST-inv MT MA)
                  (INST-in i MT))
             (INST-inv i MA))
  :hints (("Goal" :in-theory (enable MT-INST-inv INST-in)))))

;; Every instruction in a MAETT should satisfy INST-inv.
(defthm INST-inv-if-INST-in
    (implies (and (inv MT MA)
```

```
                (INST-in i MT))
           (INST-inv i MA))
  :hints (("Goal" :in-theory (enable inv))))
)

(encapsulate nil
(local
(defthm consistent-RS-entry-p-if-INST-in-help
    (implies (and (trace-consistent-RS-p trace MT MA)
                (member-equal i trace))
           (consistent-RS-entry-p i MT MA))
  :rule-classes nil))

; consistent-RS-entry-p is true for any instruction i in MT.
(defthm consistent-RS-entry-p-if-INST-in
    (implies (and (inv MT MA)
                (INST-in i MT))
           (consistent-RS-entry-p i MT MA))
  :hints (("goal" :use (:instance consistent-RS-entry-p-if-INST-in-help
                                  (trace (MT-trace MT)))
                :in-theory (enable inv consistent-RS-p
                                   INST-in)))
  :rule-classes nil)
)

;;;;;; Relations between Exceptions
(deflabel begin-exception-relations)
(defthm INST-excpt-detected-p-if-INST-fetch-error-detected-p
    (implies (INST-fetch-error-detected-p i)
           (INST-excpt-detected-p i))
  :hints (("Goal" :in-theory (enable INST-excpt-detected-p)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
           (implies (not (INST-excpt-detected-p i))
                    (not (INST-fetch-error-detected-p i))))))

(defthm INST-excpt-detected-p-if-INST-decode-error-detected-p
    (implies (INST-decode-error-detected-p i)
           (INST-excpt-detected-p i))
  :hints (("Goal" :in-theory (enable INST-excpt-detected-p)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
           (implies (not (INST-excpt-detected-p i))
                    (not (INST-decode-error-detected-p i))))))

(defthm INST-excpt-detected-p-if-INST-data-accs-error-detected-p
    (implies (INST-data-accs-error-detected-p i)
           (INST-excpt-detected-p i))
  :hints (("Goal" :in-theory (enable INST-excpt-detected-p)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
           (implies (not (INST-excpt-detected-p i))
                    (not (INST-data-accs-error-detected-p i))))))

(defthm INST-data-accs-error-detected-p-if-INST-store-accs-error-detected-p
    (implies (INST-store-accs-error-detected-p i)
           (INST-data-accs-error-detected-p i))
  :hints (("Goal" :in-theory (enable INST-data-accs-error-detected-p)))
  :rule-classes
```

```
  ((:rewrite)
   (:rewrite :corollary
             (implies (not (INST-data-accs-error-detected-p i))
                      (not (INST-store-accs-error-detected-p i))))))

(defthm INST-data-accs-error-detected-p-if-INST-load-accs-error-detected-p
    (implies (INST-load-accs-error-detected-p i)
             (INST-data-accs-error-detected-p i))
  :hints (("Goal" :in-theory (enable INST-data-accs-error-detected-p)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (not (INST-data-accs-error-detected-p i))
                      (not (INST-load-accs-error-detected-p i))))))

(defthm INST-data-accs-error-fetch-error-exclusive
    (implies (INST-data-accs-error-detected-p i)
             (not (INST-fetch-error-detected-p i)))
  :hints (("Goal" :in-theory (enable INST-function-def))))

(defthm INST-decode-error-fetch-error-exclusive
    (implies (INST-decode-error-detected-p i)
             (not (INST-fetch-error-detected-p i)))
  :hints (("Goal" :in-theory (enable INST-function-def))))

(defthm INST-data-accs-error-decode-error-exclusive
    (implies (INST-data-accs-error-detected-p i)
             (not (INST-decode-error-detected-p i)))
  :hints (("Goal" :in-theory (enable INST-function-def))))

(defthm INST-fetch-error-decode-error-exclusive
    (implies (INST-fetch-error-detected-p i)
             (not (INST-decode-error-detected-p i)))
  :hints (("Goal" :in-theory (enable INST-function-def))))

(defthm INST-decode-error-data-accs-error-exclusive
    (implies (INST-decode-error-detected-p i)
             (not (INST-data-accs-error-detected-p i)))
  :hints (("Goal" :in-theory (enable INST-function-def))))

(defthm INST-fetch-error-data-accs-error-exclusive
    (implies (INST-fetch-error-detected-p i)
             (not (INST-data-accs-error-detected-p i)))
  :hints (("Goal" :in-theory (enable INST-function-def))))

(deflabel end-exception-relations)
(deftheory exception-relations
    (set-difference-theories (universal-theory 'end-exception-relations)
                             (universal-theory 'begin-exception-relations)))
(in-theory (disable exception-relations))

;;;;; Exceptions and stages
;; Instructions with a fetch error or a decode error cannot be in the
;; execution stage.
(encapsulate nil
(local
(defthm not-fetch-error-detected-if-execute-stg-help
    (implies (and (MA-state-p MA)
                  (INST-inv i MA)
                  (INST-p i)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
```

```
                (execute-stg-p (INST-stg i)))
           (not (INST-fetch-error-detected-p i)))
  :hints (("Goal" :in-theory (enable inst-inv-def exception-relations)))))

(defthm not-fetch-error-detected-if-execute-stg
    (implies (and (inv MT MA)
                  (execute-stg-p (INST-stg i))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (INST-p i))
             (not (INST-fetch-error-detected-p i))))
)

(encapsulate nil
(local
(defthm not-decode-error-detected-if-execute-stg-help
    (implies (and (MA-state-p MA)
                  (INST-inv i MA)
                  (INST-p i)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (execute-stg-p (INST-stg i)))
             (not (INST-decode-error-detected-p i)))
  :hints (("Goal" :in-theory (enable inst-inv-def exception-relations)))))

(defthm not-decode-error-detected-if-execute-stg
    (implies (and (inv MT MA)
                  (execute-stg-p (INST-stg i))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (not (INST-decode-error-detected-p i))))
)

;;;;; Other exception related lemmas
(defthm INST-fetch-error-step-INST
    (equal (INST-fetch-error? (step-INST i MT MA sigs))
           (INST-fetch-error? i))
  :hints (("Goal" :in-theory (enable INST-fetch-error?))))

(defthm INST-decode-error-step-INST
    (equal (INST-decode-error? (step-INST i MT MA sigs))
           (INST-decode-error? i))
  :hints (("Goal" :in-theory (enable INST-decode-error?))))

(defthm INST-load-error-step-INST
    (equal (INST-load-error? (step-INST i MT MA sigs))
           (INST-load-error? i))
  :hints (("Goal" :in-theory (enable INST-load-error?))))

(defthm INST-store-error-step-INST
    (equal (INST-store-error? (step-INST i MT MA sigs))
           (INST-store-error? i))
  :hints (("Goal" :in-theory (enable INST-store-error?))))

(defthm INST-data-access-error-step-INST
    (equal (INST-data-access-error? (step-INST i MT MA sigs))
           (INST-data-access-error? i))
```

```
  :hints (("Goal" :in-theory (enable INST-data-access-error?))))

(defthm INST-fetch-error-detected-p-iff-INST-fetch-error?
    (iff (INST-fetch-error-detected-p i)
         (b1p (INST-fetch-error? i)))
  :hints (("Goal" :in-theory (enable INST-fetch-error-detected-p
                                     INST-fetch-error?)))
  :rule-classes
  ((:rewrite :corollary
             (implies (b1p (INST-fetch-error? i))
                      (INST-fetch-error-detected-p i)))
   (:rewrite :corollary
             (implies (not (b1p (INST-fetch-error? i)))
                      (not (INST-fetch-error-detected-p i))))))

(defthm INST-decode-error-detected-p-iff-INST-decode-error?
    (implies (and (b1p (INST-decode-error? i))
                  (not (b1p (INST-fetch-error? i)))
                  (not (IFU-stg-p (INST-stg i))))
             (INST-decode-error-detected-p i))
  :hints (("goal" :in-theory (enable INST-decode-error-detected-p
                                     INST-decode-error? lift-b-ops
                                     INST-opcode
                                     decode-illegal-inst? INST-su))))

(defthm not-INST-decode-error-detected-p-iff-not-INST-decode-error?
    (implies (not (b1p (INST-decode-error? i)))
             (not (INST-decode-error-detected-p i)))
  :hints (("goal" :in-theory (enable INST-decode-error-detected-p
                                     INST-decode-error? lift-b-ops
                                     INST-opcode
                                     decode-illegal-inst? INST-su))))

(defthm not-INST-decode-error-detected-p-if-IFU-stg
    (implies (IFU-stg-p (INST-stg i))
             (not (INST-decode-error-detected-p i)))
  :hints (("goal" :in-theory (enable INST-decode-error-detected-p))))

(defthm not-INST-fetch-error-if-execute-stg
    (implies (and (inv MT MA)
                  (execute-stg-p (INST-stg i))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (INST-p i))
             (equal (INST-fetch-error? i) 0))
  :hints (("goal" :in-theory (e/d (equal-b1p-converter
                                   exception-relations)
                                  (not-fetch-error-detected-if-execute-stg))
                  :use (:instance not-fetch-error-detected-if-execute-stg))))

(defthm not-INST-decode-error-if-execute-stg
    (implies (and (inv MT MA)
                  (execute-stg-p (INST-stg i))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (INST-p i))
             (equal (INST-decode-error? i) 0))
  :hints (("goal" :in-theory (e/d (equal-b1p-converter
```

```
                                  exception-relations)
                                 (not-decode-error-detected-if-execute-stg))
                 :use (:instance not-decode-error-detected-if-execute-stg))))

(encapsulate nil
(local
(defthm not-fetch-error-detected-if-wbuf-stg-help
    (implies (and (INST-p i) (MA-state-p MA)
                  (MAETT-p MT)
                  (INST-inv i MA)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (wbuf-stg-p (INST-stg i)))
             (not (INST-fetch-error-detected-p i)))
  :hints (("Goal" :in-theory (enable inst-inv-def wbuf-stg-p
                                     INST-excpt? exception-relations
                                     lift-b-ops)))))

(defthm not-fetch-error-detected-if-wbuf-stg
    (implies (and (inv MT MA)
                  (wbuf-stg-p (INST-stg i))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (INST-p i))
             (not (INST-fetch-error-detected-p i))))
)

(encapsulate nil
(local
(defthm not-decode-error-detected-if-wbuf-stg-help
    (implies (and (INST-p i) (MA-state-p MA)
                  (MAETT-p MT)
                  (INST-inv i MA)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (wbuf-stg-p (INST-stg i)))
             (not (INST-decode-error-detected-p i)))
  :hints (("Goal" :in-theory (enable inst-inv-def wbuf-stg-p
                                     INST-excpt? lift-b-ops
                                     exception-relations)))))

(defthm not-decode-error-detected-if-wbuf-stg
    (implies (and (inv MT MA)
                  (wbuf-stg-p (INST-stg i))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (not (INST-decode-error-detected-p i))))
)

(defthm not-INST-load-accs-error-detected-p-if-not-dispatched
    (implies (or (IFU-stg-p (INST-stg i))
                 (DQ-stg-p (INST-stg i)))
             (not (INST-load-accs-error-detected-p i)))
  :hints (("goal" :in-theory (enable INST-load-accs-error-detected-p))))

(defthm not-INST-store-accs-error-detected-p-if-not-dispatched
    (implies (or (IFU-stg-p (INST-stg i))
```

```
                (DQ-stg-p (INST-stg i)))
            (not (INST-store-accs-error-detected-p i)))
  :hints (("goal" :in-theory (enable INST-store-accs-error-detected-p))))

(defthm not-INST-data-accs-error-detected-p-if-not-dispatched
    (implies (or (IFU-stg-p (INST-stg i))
                (DQ-stg-p (INST-stg i)))
            (not (INST-data-accs-error-detected-p i)))
  :hints (("goal" :in-theory (enable INST-data-accs-error-detected-p))))

(defthm not-INST-data-accs-error-detected-if-INST-IU
    (implies (and (INST-p i) (b1p (INST-IU? i)))
            (not (INST-data-accs-error-detected-p i)))
  :hints (("goal" :in-theory (enable INST-data-accs-error-detected-p
                                     INST-load-accs-error-detected-p
                                     INST-store-accs-error-detected-p
                                     INST-cntlv
                                     INST-IU?))))

(defthm not-INST-data-accs-error-detected-if-INST-MU
    (implies (and (INST-p i) (b1p (INST-MU? i)))
            (not (INST-data-accs-error-detected-p i)))
  :hints (("goal" :in-theory (enable INST-data-accs-error-detected-p
                                     INST-load-accs-error-detected-p
                                     INST-store-accs-error-detected-p
                                     INST-cntlv
                                     INST-MU?))))

(defthm not-INST-data-accs-error-detected-if-INST-BU
    (implies (and (INST-p i) (b1p (INST-BU? i)))
            (not (INST-data-accs-error-detected-p i)))
  :hints (("goal" :in-theory (enable INST-data-accs-error-detected-p
                                     INST-load-accs-error-detected-p
                                     INST-store-accs-error-detected-p
                                     INST-cntlv
                                     INST-BU?))))

(defthm INST-load-accs-error-detected-p-if-complete
    (implies (and (b1p (INST-load-error? i))
                  (not (b1p (INST-fetch-error? i)))
                  (not (b1p (INST-decode-error? i)))
                  (or (complete-stg-p (INST-stg i))
                      (commit-stg-p (INST-stg i))
                      (retire-stg-p (INST-stg i))))
            (INST-load-accs-error-detected-p i))
  :hints (("goal" :in-theory (enable INST-load-accs-error-detected-p
                                     INST-load-error?
                                     lift-b-ops inst-load-addr
                                     INST-opcode INST-im INST-ra INST-rb))))

(defthm INST-store-accs-error-detected-p-if-complete
    (implies (and (b1p (INST-store-error? i))
                  (not (b1p (INST-fetch-error? i)))
                  (not (b1p (INST-decode-error? i)))
                  (or (complete-stg-p (INST-stg i))
                      (commit-stg-p (INST-stg i))
                      (retire-stg-p (INST-stg i))))
            (INST-store-accs-error-detected-p i))
  :hints (("goal" :in-theory (enable INST-store-accs-error-detected-p
                                     INST-store-error?
                                     lift-b-ops inst-store-addr
                                     INST-opcode INST-im INST-ra INST-rb))))
```

```
(defthm INST-data-accs-error-detected-p-if-complete
    (implies (and (b1p (INST-data-access-error? i))
                  (not (b1p (INST-fetch-error? i)))
                  (not (b1p (INST-decode-error? i)))
                  (or (complete-stg-p (INST-stg i))
                      (commit-stg-p (INST-stg i))
                      (retire-stg-p (INST-stg i))))
             (INST-data-accs-error-detected-p i))
  :hints (("goal" :in-theory (enable INST-data-accs-error-detected-p
                                     INST-data-access-error? lift-b-ops))))

(defthm not-INST-load-error-detected-p-if-not-INST-load-error
    (implies (not (b1p (INST-load-error? i)))
             (not (INST-load-accs-error-detected-p i)))
  :hints (("goal" :in-theory (enable INST-load-accs-error-detected-p
                                     lift-b-ops inst-load-error?
                                     INST-opcode INST-load-addr
                                     INST-function-def))))

(defthm not-INST-store-error-detected-p-if-not-INST-store-error
    (implies (not (b1p (INST-store-error? i)))
             (not (INST-store-accs-error-detected-p i)))
  :hints (("goal" :in-theory (enable INST-store-accs-error-detected-p
                                     lift-b-ops inst-store-error?
                                     INST-opcode INST-store-addr
                                     INST-function-def))))

(defthm not-INST-data-accs-error-detected-p-if-not-INST-decode-error
    (implies (not (b1p (INST-data-access-error? i)))
             (not (INST-data-accs-error-detected-p i)))
  :hints (("goal" :in-theory (enable INST-data-accs-error-detected-p
                                     lift-b-ops inst-data-access-error?))))

(defthm INST-excpt-detected-p-if-INST-excpt-completed
    (implies (and (b1p (INST-excpt? I))
                  (or (complete-stg-p (INST-stg i))
                      (commit-stg-p (INST-stg i))
                      (retire-stg-p (INST-stg i))))
             (INST-excpt-detected-p i))
  :hints (("goal" :in-theory (enable INST-excpt? INST-excpt-detected-p
                                     lift-b-ops)
                  :cases ((b1p (INST-fetch-error? i))))
          ("subgoal 2" :cases ((b1p (INST-decode-error? i))))))

(defthm not-INST-excpt-detected-if-not-INST-excpt
    (implies (and (INST-p i) (not (b1p (INST-excpt? i))))
             (not (INST-excpt-detected-p i)))
  :hints (("goal" :in-theory (enable INST-excpt? INST-excpt-detected-p
                                     lift-b-ops))))

(defthm INST-fetch-error-detected-p-step-inst
    (equal (INST-fetch-error-detected-p (step-INST i MT MA sigs))
           (INST-fetch-error-detected-p i))
  :hints (("goal" :in-theory (enable INST-fetch-error-detected-p))))

(defthm INST-data-decode-error-detected-step-INST-not-IFU
    (implies (not (IFU-stg-p (INST-stg i)))
             (equal (INST-decode-error-detected-p (step-INST i MT MA sigs))
                    (INST-decode-error-detected-p i)))
  :hints (("goal" :in-theory (enable INST-decode-error-detected-p)
                  :use (:instance (:instance INST-is-at-one-of-the-stages)))))
```

```
(defthm INST-store-accs-error-detected-p-step-inst
    (implies (INST-store-accs-error-detected-p i)
             (INST-store-accs-error-detected-p (step-INST i MT MA sigs)))
  :hints (("Goal" :in-theory (enable INST-store-accs-error-detected-p))))

(defthm INST-load-accs-error-detected-p-step-inst
    (implies (INST-load-accs-error-detected-p i)
             (INST-load-accs-error-detected-p (step-INST i MT MA sigs)))
  :hints (("Goal" :in-theory (enable INST-load-accs-error-detected-p))))

(defthm INST-data-access-error-detected-p-step-inst
    (implies (INST-data-accs-error-detected-p i)
             (INST-data-accs-error-detected-p (step-INST i MT MA sigs)))
  :hints (("Goal" :in-theory (enable INST-data-accs-error-detected-p))))

(defthm INST-excpt-detected-p-step-inst-if-not-advance
    (implies (equal (INST-stg (step-inst i MT MA sigs)) (INST-stg i))
             (equal (INST-excpt-detected-p (step-INST i MT MA sigs))
                    (INST-excpt-detected-p i)))
  :hints (("goal" :in-theory (enable INST-excpt-detected-p
                                     INST-data-accs-error-detected-p
                                     INST-store-accs-error-detected-p
                                     INST-load-accs-error-detected-p
                                     INST-decode-error-detected-p
                                     INST-fetch-error-detected-p))))

(defthm INST-excpt-step-INST
    (equal (INST-excpt? (step-INST i MT MA sigs)) (INST-excpt? i))
  :hints (("Goal" :in-theory (enable MT-def))))

(defthm INST-excpt-flags-step-inst-if-excpt-detected
    (implies (and (INST-p i) (inst-excpt-detected-p i))
             (equal (INST-excpt-flags (step-INST i MT MA sigs))
                    (INST-excpt-flags i)))
  :hints (("Goal" :in-theory (e/d (INST-excpt-flags
                                   INST-excpt-detected-p)
                                  (INST-is-at-one-of-the-stages))
                  :use (:instance INST-is-at-one-of-the-stages))))

(defthm INST-excpt-flags-step-DQ-inst
    (implies (and (DQ-stg-p (INST-stg i))
                  (DQ-stg-p (INST-stg (step-INST i MT MA sigs))))
             (equal (INST-excpt-flags (step-INST i MT MA sigs))
                    (INST-excpt-flags i)))
  :hints (("Goal" :in-theory (enable INST-excpt-flags))))

(defthm INST-excpt-flags-step-INST-if-complete
    (implies (and (inst-p i) (INST-in i MT)
                  (complete-stg-p (INST-stg i)))
             (equal (INST-excpt-flags (step-INST i MT MA sigs))
                    (INST-excpt-flags i)))
  :Hints (("goal" :in-theory (enable INST-excpt-flags
                                     INST-data-accs-error-detected-p
                                     INST-load-accs-error-detected-p
                                     INST-store-accs-error-detected-p))))

; INST-context-sync? does not change after step-INST.
(defthm inst-context-sync-step-INST
    (equal (inst-context-sync? (step-INST i MT MA sigs))
           (inst-context-sync? i))
  :hints (("goal" :in-theory (enable inst-context-sync? lift-b-ops))))
```

```
(defthm INST-wrong-branch-step-INST-if-IFU
    (implies (and (INST-p i) (MA-state-p MA) (MAETT-p MT)
                  (MA-input-p sigs) (IFU-stg-p (INST-stg i))
                  (or (not (b1p (IFU-branch-predict? (MA-IFU MA) MA sigs)))
                      (b1p (DQ-full? (MA-DQ MA)))))
             (equal (INST-wrong-branch? (step-INST i MT MA sigs))
                    (INST-wrong-branch? i)))
  :Hints (("goal" :in-theory (enable INST-wrong-branch?
                                     SIMPLIFY-BIT-FUNCTIONS-2
                                     lift-b-ops))))

(defthm inst-wrong-branch-step-INST-if-not-IFU
    (implies (and (MA-state-p MA) (MAETT-p MT) (INST-p i)
                  (MA-input-p sigs)
                  (not (IFU-stg-p (INST-stg i))))
             (equal (inst-wrong-branch? (step-INST i MT MA sigs))
                    (inst-wrong-branch? i)))
  :hints (("goal" :in-theory (e/d (inst-wrong-branch?
                                   lift-b-ops
                                   equal-b1p-converter) ()))))

;;;;;; MT-init-ISA of a MAETT is the pre-ISA of the first
;;;;;; instruction in the MAETT.
(defthm INST-pre-ISA-car-MT-trace
    (implies (and (weak-inv MT)
                  (MAETT-p MT)
                  (consp (MT-trace MT)))
             (equal (INST-pre-ISA (car (MT-trace MT)))
                    (MT-init-ISA MT)))
  :hints (("goal" :in-theory (enable weak-inv inv
                                     ISA-step-chain-p))))

(encapsulate nil
(local
 (defthm not-inst-specultv-INST-in-if-committed-help
     (implies (and (trace-no-specultv-commit-p trace)
                   (member-equal i trace)
                   (INST-listp trace)
                   (INST-p i)
                   (committed-p i))
              (equal (inst-specultv? i) 0))
   :hints (("goal" :in-theory (enable zbp committed-p)))))

;;;;;; Lemmas about MT-DQ-len
(deflabel begin-MT-DQ-len-lemmas)
(defthm mt-dq-len-0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))))
             (equal (MT-DQ-len MT) 0))
  :hints (("goal" :in-theory (enable inv misc-inv
                                     correct-entries-in-dq-p))))

(defthm mt-dq-len-1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))
                  (not (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))))
             (equal (MT-DQ-len MT) 1))
  :hints (("goal" :in-theory (enable inv misc-inv
                                     correct-entries-in-dq-p))))
```

```
(defthm mt-dq-len-2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))
                  (not (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))))
             (equal (MT-DQ-len MT) 2))
  :hints (("goal" :in-theory (enable inv misc-inv
                                     correct-entries-in-dq-p))))

(defthm mt-dq-len-3
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))
                  (not (b1p (DE-valid? (DQ-DE3 (MA-DQ MA))))))
             (equal (MT-DQ-len MT) 3))
  :hints (("goal" :in-theory (enable inv misc-inv
                                     correct-entries-in-dq-p))))

(defthm mt-dq-len-4
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (DE-valid? (DQ-DE3 (MA-DQ MA)))))
             (equal (MT-DQ-len MT) 4))
  :hints (("goal" :in-theory (enable inv misc-inv
                                     correct-entries-in-dq-p))))

(defthm mt-dq-len-ge-1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))
             (>= (MT-DQ-len MT) 1))
  :hints (("goal" :in-theory (enable inv misc-inv
                                     correct-entries-in-dq-p)))
  :rule-classes :linear)

(defthm mt-dq-len-ge-2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (DE-valid? (DQ-DE1 (MA-DQ MA)))))
             (>= (MT-DQ-len MT) 2))
  :hints (("goal" :in-theory (enable inv misc-inv
                                     correct-entries-in-dq-p)))
  :rule-classes :linear)

(defthm mt-dq-len-lt-2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))))
             (< (MT-DQ-len MT) 2))
  :hints (("goal" :in-theory (enable inv misc-inv
                                     correct-entries-in-dq-p)))
  :rule-classes :linear)

(defthm mt-dq-len-ge-3
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (DE-valid? (DQ-DE2 (MA-DQ MA)))))
             (>= (MT-DQ-len MT) 3))
  :hints (("goal" :in-theory (enable inv misc-inv
                                     correct-entries-in-dq-p)))
  :rule-classes :linear)
```

```
(defthm mt-dq-len-lt-3
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))))
             (< (MT-DQ-len MT) 3))
  :hints (("goal" :in-theory (enable inv misc-inv
                                     correct-entries-in-dq-p)))
  :rule-classes :linear)

(defthm mt-dq-len-lt-4
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (DE-valid? (DQ-DE3 (MA-DQ MA))))))
             (< (MT-DQ-len MT) 4))
  :hints (("goal" :in-theory (enable inv misc-inv
                                     correct-entries-in-dq-p)))
  :rule-classes :linear)
(deflabel end-MT-DQ-len-lemmas)
(deftheory MT-DQ-len-lemmas
    (set-difference-theories (universal-theory 'end-MT-DQ-len-lemmas)
                             (universal-theory 'begin-MT-DQ-len-lemmas)))
(defthm MT-DQ-len-le-4
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA))
             (<= (MT-DQ-len MT) 4))
  :hints (("Goal" :in-theory (enable inv misc-inv)))
  :rule-classes :linear)

;;;;;; INST-specultv? is 0 if the instruction is committed.
(defthm not-inst-specultv-INST-in-if-committed
    (implies (and (committed-p i)
                  (inv MT MA)
                  (INST-in i MT)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (INST-p i))
             (equal (inst-specultv? i) 0))
  :hints (("Goal" :in-theory (enable weak-inv inv
                                     NO-SPECULTV-COMMIT-P
                                     INST-in))))
)
(in-theory (disable not-inst-specultv-INST-in-if-committed))

(encapsulate nil
(local
(defthm INST-exintr-INST-in-if-not-retired-help
    (implies (and (trace-correct-exintr-p trace)
                  (INST-listp trace)
                  (INST-p i)
                  (member-equal i trace)
                  (not (retire-stg-p (INST-stg i))))
             (equal (INST-exintr? i) 0))
  :hints (("Goal" :in-theory (enable b1p zbp)))))

; INST-exintr? flag is off for any instruction that are not retired.
(defthm INST-exintr-INST-in-if-not-retired
    (implies (and (inv MT MA)
                  (not (retire-stg-p (INST-stg i)))
                  (INST-in i MT)
                  (MAETT-p MT)
                  (MA-state-p MA)
```

```
                (INST-p i))
            (equal (INST-exintr? i) 0))
  :hints (("Goal" :in-theory (enable weak-inv
                                     inv
                                     INST-in correct-exintr-p))))
)
(in-theory (disable INST-exintr-INST-in-if-not-retired))

;;;;;  Lemmas about INST-word
(defthm read-mem-Inst-word
    (equal (READ-MEM (ISA-PC (INST-PRE-ISA i))
                     (ISA-MEM (INST-PRE-ISA i)))
           (Inst-word i))
  :hints (("Goal" :in-theory (enable INST-word INST-in INST-pc INST-mem))))

;;;;;; Miscellaneous lemmas about INST-pre-ISA and INST-post-ISA.
;
; Relations between INST-post-ISA and INST-pre-ISA can be given
; by ISA-step.
; Note  We are not sure this rule is more useful than harmful.
(encapsulate nil
(local
 (defthm INST-post-ISA-INST-in-help
     (implies (and (ISA-chained-trace-p trace pre)
                   (member-equal i trace)
                   (INST-listp trace)
                   (INST-p i))
              (equal (INST-post-ISA i)
                     (ISA-step (INST-pre-ISA i)
                               (ISA-input (INST-exintr? i)))))))

(defthm INST-post-ISA-INST-in
    (implies (and (weak-inv MT)
                  (INST-in i MT)
                  (MAETT-p MT)
                  (INST-p i))
             (equal (INST-post-ISA i)
                    (ISA-step (INST-pre-ISA i)
                              (ISA-input (INST-exintr? i)))))
  :hints (("Goal" :in-theory (enable weak-inv inv
                                     ISA-step-chain-p
                                     INST-in ISA-chained-trace-p))))
)

(defthm INST-post-ISA-car-subtrace
    (implies (and (weak-inv MT)
                  (subtrace-p trace MT)
                  (MAETT-p MT)
                  (consp trace)
                  (INST-listp trace))
             (equal (INST-post-ISA (car trace))
                    (ISA-step (INST-pre-ISA (car trace))
                              (ISA-input (INST-exintr? (car trace)))))))
(encapsulate nil
(local
(defthm INST-pre-ISA-cadr-subtrace-help
    (implies (and (ISA-chained-trace-p trace pre)
                  (tail-p sub trace)
                  (INST-listp sub)
                  (INST-listp trace)
                  (consp sub)
                  (consp (cdr sub)))
```

```
            (equal (INST-pre-ISA (cadr sub))
                   (ISA-step (INST-pre-ISA (car sub))
                             (ISA-input (INST-exintr? (car sub))))))))

; The relations between INST-pre-ISA's of an instruction and its
; immediately following instruction can be specified by ISA-step.
(defthm INST-pre-ISA-cadr-subtrace
    (implies (and (weak-inv MT)
                  (subtrace-p trace MT)
                  (MAETT-p MT)
                  (INST-listp trace)
                  (consp trace)
                  (consp (cdr trace)))
             (equal (INST-pre-ISA (cadr trace))
                    (ISA-step (INST-pre-ISA (car trace))
                              (ISA-input (INST-exintr? (car trace))))))
  :hints (("Goal" :in-theory (enable weak-inv inv
                                     ISA-step-chain-p subtrace-p))))
)

;;;;; Relations between ISA predicate and INST predicates
(defthm ISA-store-inst-p-INST-pre-ISA
    (implies (and (weak-inv MT)
                  (INST-p i)
                  (INST-in i MT))
             (iff (ISA-store-inst-p (INST-pre-ISA i))
                  (b1p (INST-store? i))))
  :hints (("goal" :in-theory (enable INST-function-def lift-b-ops
                                     ISA-store-inst-p
                                     store-inst-p
                                     decode rdb logbit*))))

(defthm ISA-store-addr-INST-pre-ISA
    (implies (and (weak-inv MT)
                  (INST-p i)
                  (INST-in i MT))
             (equal (ISA-store-addr (INST-pre-ISA i))
                    (INST-store-addr i)))
  :hints (("goal" :in-theory (enable INST-function-def lift-b-ops
                                     ISA-store-addr
                                     store-inst-p
                                     decode rdb logbit*))))
(defthm ISA-excpt-p-INST-pre-ISA
    (implies (and (weak-inv MT)
                  (INST-p i)
                  (INST-in i MT))
             (iff (ISA-excpt-p (INST-pre-ISA i))
                  (b1p (INST-excpt? i))))
  :hints (("goal" :in-theory (enable INST-function-def lift-b-ops
                                     ISA-excpt-p
                                     decode rdb logbit*
                                     INST-function-def
                                     ISA-functions
                                     MA-def))))

(encapsulate nil
(local
(defthm no-tag-conflict-at-if-no-tag-conflict-under
    (implies (and (no-tag-conflict-under upper MT MA)
                  (rob-index-p rix)
                  (integerp upper)
                  (< rix upper))
```

```
            (no-tag-conflict-at rix MT MA))))

(local
(defthm no-tag-conflict-at-if-no-tag-conflict
    (implies (and (no-tag-conflict MT MA)
                  (rob-index-p rix))
             (no-tag-conflict-at rix MT MA))
  :hints (("Goal" :in-theory (enable no-tag-conflict rob-index-p)))))

;  ROB conflict should not appear at any ROB index.
(defthm no-tag-conflict-at-all-rix
    (implies (and (inv MT MA)
                  (rob-index-p rix))
             (no-tag-conflict-at rix MT MA))
  :hints (("Goal" :in-theory (enable weak-inv inv))))
) ;encapsulate

;;;;;; Lemmas about subtraces and trace predicates.
(encapsulate nil
(local
(defthm trace-final-ISA-subtrace-help
    (implies (and (consp trace1) (tail-p trace1 trace2))
             (equal (equal (trace-final-ISA trace1 pre1)
                           (trace-final-ISA trace2 pre2))
                    t))))

(defthm trace-final-ISA-subtrace
    (implies (and (consp trace) (subtrace-p trace MT))
             (equal (trace-final-ISA trace pre)
                    (MT-final-ISA MT)))
  :hints (("Goal" :in-theory (enable MT-final-ISA subtrace-p)))
  :rule-classes nil)
)

(encapsulate nil
(local
(defthm trace-specultv-subtrace-help
    (implies (and (tail-p sub trace)
                  (not (b1p (trace-specultv? trace))))
             (equal (trace-specultv? sub) 0))
  :hints (("goal" :in-theory (enable b1p zbp)))))

(defthm trace-specultv-subtrace
    (implies (and (inv MT MA)
                  (subtrace-p sub MT)
                  (not (b1p (MT-specultv? MT))))
             (equal (trace-specultv? sub) 0))
  :hints (("Goal" :in-theory (enable weak-inv inv
                                     subtrace-p MT-specultv?))))
)

(encapsulate nil
(local
(defthm ISA-chained-trace-p-subtrace-help
    (implies (and (tail-p sub trace)
                  (INST-listp trace)
                  (INST-listp sub)
                  (ISA-chained-trace-p trace (INST-pre-ISA (car trace)))
                  (consp sub))
             (ISA-chained-trace-p sub (INST-pre-ISA (car sub))))
  :hints ((when-found (ISA-CHAINED-TRACE-P (CDR TRACE)
                                           (INST-POST-ISA (CAR TRACE)))
```

```
                    (:expand (ISA-CHAINED-TRACE-P (CDR TRACE)
                                        (INST-POST-ISA (CAR TRACE))))))))

(defthm ISA-chained-trace-p-subtrace
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (INST-listp trace)
                  (consp trace)
                  (subtrace-p trace MT))
             (ISA-chained-trace-p trace (INST-pre-ISA (car trace))))
  :hints (("Goal" :in-theory (e/d (subtrace-p inv weak-inv
                                              ISA-step-chain-p))
                  :cases ((consp (MT-trace MT))))))
)

(defthm ISA-chained-trace-p-INST-pre-ISA-of-car
    (implies (ISA-chained-trace-p trace whatever)
             (ISA-chained-trace-p trace (INST-pre-ISA (car trace)))))

(encapsulate nil
(local
(defthm correct-exintr-p-subtrace-help
    (implies (and (trace-correct-exintr-p trace)
                  (tail-p sub trace))
             (trace-correct-exintr-p sub)))
)
(defthm correct-exintr-p-subtrace
    (implies (and (inv MT MA)
                  (subtrace-p sub MT))
             (trace-correct-exintr-p sub))
  :hints (("Goal" :in-theory (enable inv
                                     weak-inv correct-exintr-p subtrace-p))))
)

(encapsulate nil
(local
(defthm no-specultv-commit-p-subtrace-help
    (implies (and (trace-no-specultv-commit-p trace)
                  (tail-p sub trace))
             (trace-no-specultv-commit-p sub))))

(defthm no-specultv-commit-p-subtrace
    (implies (and (inv MT MA)
                  (subtrace-p sub MT))
             (trace-no-specultv-commit-p sub))
  :hints (("Goal" :in-theory (enable weak-inv inv
                                     no-specultv-commit-p
                                     subtrace-p))))
)

(defthm trace-INST-inv-subtrace
    (implies (and (inv MT MA)
                  (subtrace-p sub MT))
             (trace-INST-inv sub MA)))

(encapsulate nil
(local
(defthm in-order-trace-p-subtrace-help
    (implies (and (in-order-trace-p trace)
                  (tail-p sub trace))
             (in-order-trace-p sub))))
```

```
(defthm in-order-trace-p-subtrace
    (implies (and (inv MT MA)
                  (subtrace-p sub MT))
             (in-order-trace-p sub))
  :hints (("goal" :in-theory (enable weak-inv inv
                                     in-order-dispatch-commit-p
                                     subtrace-p))))
)

(encapsulate nil
(local
(defthm in-order-trace-p-subtrace-help
    (implies (and (in-order-trace-p trace)
                  (tail-p sub trace))
             (in-order-trace-p sub))))

(defthm no-stage-conflict-subtrace
    (implies (and (inv MT MA)
                  (subtrace-p sub MT))
             (in-order-trace-p sub))
  :hints (("goal" :in-theory (enable weak-inv inv
                                     in-order-dispatch-commit-p
                                     subtrace-p))))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Lemmas related to INST-at-stg
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; INST-at-stg is INST-p
(encapsulate nil
(local
(defthm INST-p-INST-at-stg-in-trace
    (implies (and (uniq-INST-at-stg-in-trace stg trace)
                  (INST-listp trace))
             (INST-p (INST-at-stg-in-trace stg trace)))))

(defthm INST-p-INST-at-stg
    (implies (and (uniq-INST-at-stg stg MT)
                  (MAETT-p MT))
             (INST-p (INST-at-stg stg MT)))
  :hints (("Goal" :in-theory (enable uniq-INST-at-stg INST-at-stg))))
)

(encapsulate nil
(local
(defthm INST-in-INST-at-stg-help
    (implies (uniq-INST-at-stg-in-trace stg trace)
             (member-equal (INST-at-stg-in-trace stg trace) trace))))

;;; INST-at-stg belongs to a MAETT.
(defthm INST-in-INST-at-stg
    (implies (uniq-INST-at-stg stg MT)
             (INST-in (INST-at-stg stg MT) MT))
  :hints (("goal" :in-theory (enable INST-in INST-at-stg uniq-INST-at-stg))))
)

(defthm uniq-INST-at-IFU-if-IFU-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (IFU-valid? (MA-IFU MA))))
             (uniq-INST-at-stg '(IFU) MT))
```

```
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-IFU-stg-conflict))))

(encapsulate nil
(local
(defthm INST-p-INST-at-stgs-in-trace
    (implies (and (uniq-INST-at-stgs-in-trace stgs trace)
                  (INST-listp trace))
             (INST-p (INST-at-stgs-in-trace stgs trace)))))

(defthm INST-p-INST-at-stgs
    (implies (and (uniq-INST-at-stgs stgs MT)
                  (MAETT-p MT))
             (INST-p (INST-at-stgs stgs MT)))
  :hints (("Goal" :in-theory (enable uniq-INST-at-stgs INST-at-stgs))))
)

(encapsulate nil
(local
(defthm INST-in-INST-at-stgs-help
    (implies (uniq-INST-at-stgs-in-trace stgs trace)
             (member-equal (INST-at-stgs-in-trace stgs trace) trace))))

;;; INST-at-stgs belongs to a MAETT.
(defthm INST-in-INST-at-stgs
    (implies (uniq-INST-at-stgs stgs MT)
             (INST-in (INST-at-stgs stgs MT) MT))
  :hints (("goal" :in-theory (enable INST-in INST-at-stgs uniq-INST-at-stgs))))
)

(encapsulate nil
(local
(defthm equal-INST-stg-INST-at-stgs-in-trace
    (implies (and (stage-p stg)
                  (not (member-equal stg stgs)))
             (not (equal (INST-stg (INST-at-stgs-in-trace stgs trace)) stg)))
  :rule-classes nil))

; The stage of the instruction returned by INST-at-stgs is one of the
; first argument.
(defthm equal-INST-stg-INST-at-stgs
    (implies (and (stage-p stg)
                  (not (member-equal stg stgs)))
             (not (equal (INST-stg (INST-at-stgs stgs MT)) stg)))
  :hints (("goal" :In-theory (enable INST-at-stgs)
                  :use (:instance equal-INST-stg-INST-at-stgs-in-trace
                                  (trace (MT-trace MT))))))
)

(encapsulate nil
(local
(defthm uniq-inst-at-stg-no-inst-at-stg-exclusive-help
    (implies (uniq-inst-at-stg-in-trace rix trace)
             (not (no-inst-at-stg-in-trace rix trace)))))

(defthm uniq-inst-at-stg-no-inst-at-stg-exclusive
    (implies (uniq-inst-at-stg rix MT)
             (not (no-inst-at-stg rix MT)))
  :hints (("goal" :in-theory (enable no-inst-at-stg uniq-inst-at-stg)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
```

```
            (implies (no-inst-at-stg rix MT)
                     (not (uniq-inst-at-stg rix MT))))))
)

(encapsulate nil
(local
(defthm no-inst-at-stgs-in-trace-no-inst-at-stg-in-trace-1
    (implies (and (consp stgs)
                  (no-inst-at-stgs-in-trace stgs trace))
             (no-inst-at-stg-in-trace (car stgs) trace))))

(local
(defthm no-inst-at-stgs-in-trace-no-inst-at-stg-in-trace-2
    (implies (and (consp stgs)
                  (no-inst-at-stgs-in-trace stgs trace))
             (no-inst-at-stgs-in-trace (cdr stgs) trace))))

(local
(defthm not-uniq-inst-at-stg-in-trace-if-no-inst-at-stg-in-trace
    (implies (and (consp stgs)
                  (no-inst-at-stg-in-trace stg trace))
             (not (uniq-inst-at-stg-in-trace stg trace)))))

(local
(defthm uniq-inst-at-stgs-in-trace-uniq-inst-at-stg-in-trace
    (implies (and (consp stgs)
                  (not (uniq-inst-at-stg-in-trace (car stgs) trace))
                  (not (uniq-inst-at-stgs-in-trace (cdr stgs) trace)))
             (not (uniq-inst-at-stgs-in-trace stgs trace)))))

(local
 (defthm uniq-inst-at-stgs-in-trace-endp
     (implies (endp stgs)
              (not (uniq-inst-at-stgs-in-trace stgs trace)))))

; Relations between uniq-inst-at-stg and uniq-inst-at-stgs.
(defthm uniq-inst-at-stgs*
    (implies (and (consp stgs) (uniq-inst-at-stgs stgs MT))
             (or (uniq-inst-at-stg (car stgs) MT)
                 (uniq-inst-at-stgs (cdr stgs) MT)))
  :hints (("goal" :in-theory (enable uniq-inst-at-stg uniq-inst-at-stgs)))
  :rule-classes nil)

(local
(defthm inst-at-stgs-inst-at-stg-help
    (implies (uniq-inst-at-stgs-in-trace stgs trace)
             (equal (inst-at-stgs-in-trace stgs trace)
                    (if (uniq-inst-at-stg-in-trace (car stgs) trace)
                        (inst-at-stg-in-trace (car stgs) trace)
                        (inst-at-stgs-in-trace (cdr stgs) trace))))))

; Relations between inst-at-stgs and inst-at-stg.
(defthm inst-at-stgs*
    (implies (uniq-inst-at-stgs stgs MT)
             (equal (inst-at-stgs stgs MT)
                    (if (uniq-inst-at-stg (car stgs) MT)
                        (inst-at-stg (car stgs) MT)
                        (inst-at-stgs (cdr stgs) MT))))
  :hints (("goal" :in-theory (enable uniq-inst-at-stg uniq-inst-at-stgs
                                     inst-at-stg inst-at-stgs))))

(in-theory (disable inst-at-stgs*))
```

```
(local
(defthm uniq-inst-at-stgs-singleton-help
    (equal (uniq-inst-at-stgs-in-trace (list stg) trace)
           (uniq-inst-at-stg-in-trace stg trace))))

; uniq-inst-at-stgs can be reduced to uniq-inst-at-stg if the first
; argument is a singleton of stages.
(defthm uniq-inst-at-stgs-singleton
    (equal (uniq-inst-at-stgs (list stg) MT)
           (uniq-inst-at-stg stg MT))
  :hints (("goal" :in-theory (enable uniq-inst-at-stgs uniq-inst-at-stg))))

)

(encapsulate nil
(local
(defthm no-inst-at-stgs-in-trace-remove-equal
    (implies (no-inst-at-stgs-in-trace stgs trace)
             (no-inst-at-stgs-in-trace (remove-equal stg stgs) trace))))

(local
(defthm uniq-inst-at-stgs-remove-equal-help-help
    (implies (and (member-equal stg stgs)
                  (not (no-inst-at-stg-in-trace stg trace)))
             (not (no-inst-at-stgs-in-trace stgs trace)))))

(local
(defthm uniq-inst-at-stgs-remove-equal-help
    (implies (and (uniq-inst-at-stgs-in-trace stgs trace)
                  (not (uniq-inst-at-stg-in-trace stg trace))
                  (member-equal stg stgs))
             (uniq-inst-at-stgs-in-trace (remove-equal stg stgs) trace))))

(defthm uniq-inst-at-stgs-remove-equal
    (implies (and (uniq-inst-at-stgs stgs MT)
                  (not (uniq-inst-at-stg stg MT))
                  (member-equal stg stgs))
             (uniq-inst-at-stgs (remove-equal stg stgs) MT))
  :hints (("Goal" :in-theory (enable uniq-inst-at-stgs uniq-inst-at-stg))))
)

(defthm no-INST-at-IFU-if-IFU-invalid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (IFU-valid? (MA-IFU MA)))))
             (no-INST-at-stg '(IFU) MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-IFU-stg-conflict))))

(defthm IFU-valid-if-IFU-stg-p
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (IFU-stg-p (INST-stg i))
                  (INST-p i) (INST-in i MT))
             (equal (IFU-valid? (MA-IFU MA)) 1))
  :hints (("goal" :use (:instance INST-INV-IF-INST-IN)
                  :in-theory (e/d (INST-inv-def lift-b-ops
                                                equal-b1p-converter)
                                  (INST-INV-IF-INST-IN)))))

(defthm uniq-inst-at-stg-if-DQ-DEO-valid
```

```
    (implies (and (inv MT MA)
                  (MA-state-p MA) (MAETT-p MT)
                  (b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))
             (uniq-inst-at-stg '(DQ 0) MT))
  :hints (("goal" :in-theory (enable inv NO-stage-conflict
                                     no-dq-stg-conflict
                                     lift-b-ops
                                     MA-def))))

(defthm no-inst-at-stg-if-no-DQ-DE0-valid
    (implies (and (inv MT MA)
                  (MA-state-p MA) (MAETT-p MT)
                  (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))))
             (no-inst-at-stg '(DQ 0) MT))
  :hints (("goal" :in-theory (enable inv NO-stage-conflict
                                     no-dq-stg-conflict
                                     lift-b-ops
                                     MA-def))))
(defthm uniq-inst-at-stg-if-DQ-DE1-valid
    (implies (and (inv MT MA)
                  (MA-state-p MA) (MAETT-p MT)
                  (b1p (DE-valid? (DQ-DE1 (MA-DQ MA)))))
             (uniq-inst-at-stg '(DQ 1) MT))
  :hints (("goal" :in-theory (enable inv NO-stage-conflict
                                     no-dq-stg-conflict
                                     lift-b-ops
                                     MA-def))))

(defthm no-inst-at-stg-if-no-DQ-DE1-valid
    (implies (and (inv MT MA)
                  (MA-state-p MA) (MAETT-p MT)
                  (not (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))))
             (no-inst-at-stg '(DQ 1) MT))
  :hints (("goal" :in-theory (enable inv NO-stage-conflict
                                     no-dq-stg-conflict
                                     lift-b-ops
                                     MA-def))))

(defthm uniq-inst-at-stg-if-DQ-DE2-valid
    (implies (and (inv MT MA)
                  (MA-state-p MA) (MAETT-p MT)
                  (b1p (DE-valid? (DQ-DE2 (MA-DQ MA)))))
             (uniq-inst-at-stg '(DQ 2) MT))
  :hints (("goal" :in-theory (enable inv NO-stage-conflict
                                     no-dq-stg-conflict
                                     lift-b-ops
                                     MA-def))))

(defthm no-inst-at-stg-if-no-DQ-DE2-valid
    (implies (and (inv MT MA)
                  (MA-state-p MA) (MAETT-p MT)
                  (not (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))))
             (no-inst-at-stg '(DQ 2) MT))
  :hints (("goal" :in-theory (enable inv NO-stage-conflict
                                     no-dq-stg-conflict
                                     lift-b-ops
                                     MA-def))))

(defthm uniq-inst-at-stg-if-DQ-DE3-valid
    (implies (and (inv MT MA)
                  (MA-state-p MA) (MAETT-p MT)
                  (b1p (DE-valid? (DQ-DE3 (MA-DQ MA)))))
```

```
            (uniq-inst-at-stg '(DQ 3) MT))
  :hints (("goal" :in-theory (enable inv NO-stage-conflict
                                     no-dq-stg-conflict
                                     lift-b-ops
                                     MA-def))))

(defthm no-inst-at-stg-if-no-DQ-DE3-valid
    (implies (and (inv MT MA)
                  (MA-state-p MA) (MAETT-p MT)
                  (not (b1p (DE-valid? (DQ-DE3 (MA-DQ MA))))))
             (no-inst-at-stg '(DQ 3) MT))
  :hints (("goal" :in-theory (enable inv NO-stage-conflict
                                     no-dq-stg-conflict
                                     lift-b-ops
                                     MA-def))))

(encapsulate nil
(local
 (defthm INST-stg-INST-at-stg-help
     (implies (uniq-inst-at-stg-in-trace stg trace)
              (equal (inst-stg (INST-at-stg-in-trace stg trace)) stg))))

(defthm INST-stg-INST-at-stg
    (implies (uniq-inst-at-stg stg MT)
             (equal (INST-stg (INST-at-stg stg MT)) stg))
  :hints (("Goal" :in-theory (enable uniq-inst-at-stg
                                     INST-at-stg))))
)

(encapsulate nil
(local
 (defthm not-no-INST-at-stg-INST-stg-if-member-equal
     (implies (and (member-equal i trace)
                   (equal (INST-stg i) stg))
              (not (no-INST-at-stg-in-trace stg trace)))))

(local
 (defthm INST-at-stg-INST-stg-help
     (implies (and (uniq-INST-at-stg-in-trace stg trace)
                   (member-equal i trace)
                   (equal (INST-stg i) stg))
              (equal (INST-at-stg-in-trace stg trace) i))))

(defthm INST-at-stg-INST-stg-IFU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (IFU-stg-p (INST-stg i)))
             (equal (INST-at-stg (INST-stg i) MT) i))
  :hints (("goal" :in-theory (enable inv
                                     no-stage-conflict
                                     no-IFU-stg-conflict
                                     INST-in MA-stg-def
                                     uniq-inst-at-stg
                                     no-inst-at-stg
                                     INST-at-stg))))

(defthm INST-at-stg-INST-stg-DQ
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (DQ-stg-p (INST-stg i)))
```

```
            (equal (INST-at-stg (INST-stg i) MT) i))
  :hints (("goal" :in-theory (enable inv
                                     no-stage-conflict
                                     no-DQ-stg-conflict
                                     INST-in MA-stg-def
                                     uniq-inst-at-stg
                                     no-inst-at-stg
                                     INST-at-stg))))

(defthm INST-at-stg-INST-stg-IU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (IU-stg-p (INST-stg i)))
             (equal (INST-at-stg (INST-stg i) MT) i))
  :hints (("goal" :in-theory (enable inv
                                     no-stage-conflict
                                     no-IU-stg-conflict
                                     INST-in MA-stg-def
                                     uniq-inst-at-stg
                                     no-inst-at-stg
                                     INST-at-stg))))

(defthm INST-at-stg-INST-stg-MU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (MU-stg-p (INST-stg i)))
             (equal (INST-at-stg (INST-stg i) MT) i))
  :hints (("goal" :in-theory (enable inv
                                     no-stage-conflict
                                     no-MU-stg-conflict
                                     INST-in MA-stg-def
                                     uniq-inst-at-stg
                                     no-inst-at-stg
                                     INST-at-stg))))

(defthm INST-at-stg-INST-stg-BU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (BU-stg-p (INST-stg i)))
             (equal (INST-at-stg (INST-stg i) MT) i))
  :hints (("goal" :in-theory (enable inv
                                     no-stage-conflict
                                     no-BU-stg-conflict
                                     INST-in MA-stg-def
                                     uniq-inst-at-stg
                                     no-inst-at-stg
                                     INST-at-stg))))
)

(defun all-LSU-stg-p (stgs)
  (if (endp stgs)
      T
      (and (LSU-stg-p (car stgs))
           (all-LSU-stg-p (cdr stgs)))))

(defun all-execute-stg-p (stgs)
  (if (endp stgs)
      T
      (and (execute-stg-p (car stgs))
```

```
          (all-execute-stg-p (cdr stgs)))))

(defun all-complete-stg-p (stgs)
  (if (endp stgs)
      T
      (and (complete-stg-p (car stgs))
           (all-complete-stg-p (cdr stgs)))))

(defun all-commit-stg-p (stgs)
  (if (endp stgs)
      T
      (and (commit-stg-p (car stgs))
           (all-commit-stg-p (cdr stgs)))))

(defun all-wbuf-stg-p (stgs)
  (if (endp stgs) t
      (and (wbuf-stg-p (car stgs))
           (all-wbuf-stg-p (cdr stgs)))))

(defun all-wbuf0-stg-p (stgs)
  (if (endp stgs) t
      (and (wbuf0-stg-p (car stgs))
           (all-wbuf0-stg-p (cdr stgs)))))

(defun all-wbuf1-stg-p (stgs)
  (if (endp stgs) t
      (and (wbuf1-stg-p (car stgs))
           (all-wbuf1-stg-p (cdr stgs)))))

(encapsulate nil
(local
(defthm LSU-stg-p-INST-at-stgs-in-trace
    (implies (and (all-LSU-stg-p stgs)
                  (uniq-inst-at-stgs-in-trace stgs trace))
             (LSU-stg-p (INST-stg (INST-at-stgs-in-trace stgs trace))))))

(defthm LSU-stg-p-INST-at-stgs
    (implies (and (all-LSU-stg-p stgs)
                  (uniq-inst-at-stgs stgs MT))
             (LSU-stg-p (INST-stg (INST-at-stgs stgs MT))))
  :hints (("goal" :in-theory (enable INST-at-stgs uniq-inst-at-stgs))))
)

(encapsulate nil
(local
 (defthm execute-stg-p-INST-at-stgs-in-trace
     (implies (and (all-execute-stg-p stgs)
                   (uniq-inst-at-stgs-in-trace stgs trace))
              (execute-stg-p (INST-stg (INST-at-stgs-in-trace stgs trace))))))

(defthm execute-stg-p-INST-at-stgs
    (implies (and (all-execute-stg-p stgs)
                  (uniq-inst-at-stgs stgs MT))
             (execute-stg-p (INST-stg (INST-at-stgs stgs MT))))
  :hints (("goal" :in-theory (enable INST-at-stgs uniq-inst-at-stgs))))
)

(encapsulate nil
(local
(defthm complete-stg-p-INST-at-stgs-in-trace
    (implies (and (all-complete-stg-p stgs)
                  (uniq-inst-at-stgs-in-trace stgs trace))
```

```
            (complete-stg-p (INST-stg (INST-at-stgs-in-trace stgs trace))))))

(defthm complete-stg-p-INST-at-stgs
    (implies (and (all-complete-stg-p stgs)
                  (uniq-inst-at-stgs stgs MT))
             (complete-stg-p (INST-stg (INST-at-stgs stgs MT))))
  :hints (("goal" :in-theory (enable uniq-inst-at-stgs INST-at-stgs))))

)

(encapsulate nil
(local
(defthm commit-stg-p-INST-at-stgs-in-trace
    (implies (and (all-commit-stg-p stgs)
                  (uniq-inst-at-stgs-in-trace stgs trace))
             (commit-stg-p (INST-stg (INST-at-stgs-in-trace stgs trace))))))

(defthm commit-stg-p-INST-at-stgs
    (implies (and (all-commit-stg-p stgs)
                  (uniq-inst-at-stgs stgs MT))
             (commit-stg-p (INST-stg (INST-at-stgs stgs MT))))
  :hints (("goal" :in-theory (enable uniq-inst-at-stgs INST-at-stgs))))
)

(encapsulate nil
(local
(defthm wbuf-stg-p-INST-at-stgs-help
    (implies (and (all-wbuf-stg-p stgs)
                  (uniq-inst-at-stgs-in-trace stgs trace))
             (wbuf-stg-p (inst-stg (INST-at-stgs-in-trace stgs trace))))))

(defthm wbuf-stg-p-INST-at-stgs
    (implies (and (all-wbuf-stg-p stgs)
                  (uniq-inst-at-stgs stgs MT))
             (wbuf-stg-p (inst-stg (INST-at-stgs stgs MT))))
  :hints (("goal" :in-theory (enable INST-at-stgs uniq-inst-at-stgs))))
)

(encapsulate nil
(local
(defthm wbuf0-stg-p-INST-at-stgs-help
    (implies (and (all-wbuf0-stg-p stgs)
                  (uniq-inst-at-stgs-in-trace stgs trace))
             (wbuf0-stg-p (inst-stg (INST-at-stgs-in-trace stgs trace))))))

(defthm wbuf0-stg-p-INST-at-stgs
    (implies (and (all-wbuf0-stg-p stgs)
                  (uniq-inst-at-stgs stgs MT))
             (wbuf0-stg-p (inst-stg (INST-at-stgs stgs MT))))
  :hints (("goal" :in-theory (enable INST-at-stgs uniq-inst-at-stgs))))
)

(encapsulate nil
(local
(defthm wbuf1-stg-p-INST-at-stgs-help
    (implies (and (all-wbuf1-stg-p stgs)
                  (uniq-inst-at-stgs-in-trace stgs trace))
             (wbuf1-stg-p (inst-stg (INST-at-stgs-in-trace stgs trace))))))

(defthm wbuf1-stg-p-INST-at-stgs
    (implies (and (all-wbuf1-stg-p stgs)
                  (uniq-inst-at-stgs stgs MT))
```

```
            (wbuf1-stg-p (inst-stg (INST-at-stgs stgs MT))))
  :hints (("goal" :in-theory (enable INST-at-stgs uniq-inst-at-stgs))))
)

(encapsulate nil
(local
 (defthm strong-no-inst-at-stg-INST-stg-help
     (implies (and (member-equal i trace)
                   (equal (INST-stg i) stg))
              (not (no-INST-at-stg-in-trace stg trace)))))

; If i is in MT, and i's stage is stg, then (no-INST-at-stg stg MT) is false.
(defthm strong-no-inst-at-stg-INST-stg
    (implies (and (INST-in i MT)
                  (equal (INST-stg i) stg))
             (not (no-INST-at-stg stg MT)))
  :Hints (("goal" :in-theory (enable INST-in no-INST-at-stg))))

(local
 (defthm INST-at-stg-INST-stg-help
     (implies (and (uniq-INST-at-stg-in-trace stg trace)
                   (member-equal i trace)
                   (equal (INST-stg i) stg))
              (equal (INST-at-stg-in-trace stg trace) i))))

; If i is in MT, then (INST-at-stg (INST-stg i) MT) = i.
(defthm strong-INST-at-stg-inst-stg
    (implies (and (uniq-INST-at-stg stg MT)
                  (INST-in i MT)
                  (equal (INST-stg i) stg))
             (equal (INST-at-stg stg MT) i))
  :hints (("goal" :in-theory (enable INST-at-stg uniq-INST-at-stg INST-in))))
)

(defthm INST-at-stg-inst-stg
    (implies (and (INST-in i MT) (uniq-INST-at-stg (INST-stg i) MT))
             (equal (INST-at-stg (INST-stg i) MT) i))
  :hints (("goal" :restrict ((strong-INST-at-stg-inst-stg
                              ((i i)))))))


(encapsulate nil
(local
(defthm not-no-inst-at-stgs-inst-stg-if-member-equal
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (member-equal i trace)
                  (member-equal (INST-stg i) stgs))
             (not (no-inst-at-stgs-in-trace stgs trace)))))

; If a MAETT contains an instruction whose stage is a member of stgs,
; (non-inst-at-stgs stgs MT) is false.
(defthm not-no-inst-at-stgs-inst-stg-if-INST-in
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (member-equal (INST-stg i) stgs))
             (not (no-inst-at-stgs stgs MT)))
  :hints (("goal" :in-theory (enable no-inst-at-stgs INST-in) )))
)

(encapsulate nil
```

```
(local
(defthm uniq-inst-at-stg-if-uniq-inst-at-stgs-help-help
    (implies (and (member-equal i trace)
                  (member-equal (inst-stg i) stgs))
             (not (no-inst-at-stgs-in-trace stgs trace)))))

(local
(defthm uniq-inst-at-stg-if-uniq-inst-at-stgs-help
    (implies (and (member-equal i trace)
                  (uniq-inst-at-stgs-in-trace stgs trace)
                  (equal (INST-stg i) stg)
                  (member-equal stg stgs))
             (uniq-inst-at-stg-in-trace stg trace))))

(defthm uniq-inst-at-stg-if-uniq-inst-at-stgs
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (uniq-inst-at-stgs stgs MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg I) stg)
                  (member-equal stg stgs))
             (uniq-inst-at-stg stg MT))
  :hints (("goal" :in-theory (enable uniq-inst-at-stg uniq-inst-at-stgs
                                     INST-in))))
)

(deftheory strong-inst-at-stg-theory
    '(strong-no-inst-at-stg-INST-stg uniq-inst-at-stg-if-uniq-inst-at-stgs
      strong-INST-at-stg-inst-stg not-no-inst-at-stgs-inst-stg-if-INST-in))
(in-theory (disable strong-inst-at-stg-theory))

(encapsulate nil
(local
(defthm INST-at-stg-INST-stg-LSU-RS0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (equal (INST-stg I) '(LSU RS0)))
             (equal (INST-at-stg '(LSU RS0) MT) i))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     strong-inst-at-stg-theory
                                     no-LSU-stg-conflict)))))

(local
(defthm INST-at-stg-INST-stg-LSU-RS1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (equal (INST-stg I) '(LSU RS1)))
             (equal (INST-at-stg '(LSU RS1) MT) i))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     strong-inst-at-stg-theory
                                     no-LSU-stg-conflict)))))

(local
(defthm INST-at-stg-INST-stg-LSU-RBUF
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (equal (INST-stg I) '(LSU RBUF)))
             (equal (INST-at-stg '(LSU RBUF) MT) i))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
```

```
                                    strong-inst-at-stg-theory
                                    no-LSU-stg-conflict)))))

(local
(defthm INST-at-stg-INST-stg-LSU-wbuf0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (equal (INST-stg I) '(LSU wbuf0)))
             (equal (INST-at-stg '(LSU wbuf0) MT) i))
  :hints (("goal" :in-theory (e/d (no-stage-conflict
                                   strong-inst-at-stg-theory
                                   no-LSU-stg-conflict)
                                  (no-stage-conflict-p-forward))
                  :use ((:instance no-stage-conflict-p-forward (MA MA)))
                  :restrict ((uniq-inst-at-stg-if-uniq-inst-at-stgs
                              ((stgs '((LSU WBUF0)
                                       (LSU WBUF0 LCH)
                                       (COMPLETE WBUF0)
                                       (COMMIT WBUF0)))))))))

(local
(defthm INST-at-stg-INST-stg-LSU-wbuf0-lch
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (equal (INST-stg I) '(LSU wbuf0 lch)))
             (equal (INST-at-stg '(LSU wbuf0 lch) MT) i))
  :hints (("goal" :in-theory (e/d (no-stage-conflict
                                   strong-inst-at-stg-theory
                                   no-LSU-stg-conflict)
                                  (no-stage-conflict-p-forward))
                  :use ((:instance no-stage-conflict-p-forward (MA MA)))
                  :restrict ((uniq-inst-at-stg-if-uniq-inst-at-stgs
                              ((stgs '((LSU WBUF0)
                                       (LSU WBUF0 LCH)
                                       (COMPLETE WBUF0)
                                       (COMMIT WBUF0)))))))))

(local
(defthm INST-at-stg-INST-stg-LSU-wbuf1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (equal (INST-stg I) '(LSU wbuf1)))
             (equal (INST-at-stg '(LSU wbuf1) MT) i))
  :hints (("goal" :in-theory (e/d (no-stage-conflict
                                   strong-inst-at-stg-theory
                                   no-LSU-stg-conflict)
                                  (no-stage-conflict-p-forward))
                  :use ((:instance no-stage-conflict-p-forward (MA MA)))
                  :restrict ((uniq-inst-at-stg-if-uniq-inst-at-stgs
                              ((stgs '((LSU WBUF1)
                                       (LSU WBUF1 LCH)
                                       (COMPLETE WBUF1)
                                       (COMMIT WBUF1)))))))))

(local
(defthm INST-at-stg-INST-stg-LSU-wbuf1-lch
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
```

```
                (equal (INST-stg I) '(LSU wbuf1 lch)))
           (equal (INST-at-stg '(LSU wbuf1 lch) MT) i))
  :hints (("goal" :in-theory (e/d (no-stage-conflict
                                   strong-inst-at-stg-theory
                                   no-LSU-stg-conflict)
                                  (no-stage-conflict-p-forward))
                :use ((:instance no-stage-conflict-p-forward (MA MA)))
                :restrict ((uniq-inst-at-stg-if-uniq-inst-at-stgs
                            ((stgs '((LSU WBUF1)
                                     (LSU WBUF1 LCH)
                                     (COMPLETE WBUF1)
                                     (COMMIT WBUF1)))))))))

(local
(defthm INST-at-stg-INST-stg-LSU-lch
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (INST-in i MT)
                (equal (INST-stg I) '(LSU lch)))
           (equal (INST-at-stg '(LSU lch) MT) i))
  :hints (("goal" :in-theory (e/d (no-stage-conflict
                                   strong-inst-at-stg-theory
                                   no-LSU-stg-conflict)
                                  (no-stage-conflict-p-forward))
                :use ((:instance no-stage-conflict-p-forward (MA MA)))
                :restrict ((uniq-inst-at-stg-if-uniq-inst-at-stgs
                            ((stgs '((LSU LCH)
                                     (LSU WBUF0 LCH)
                                     (LSU WBUF1 LCH)))))))))

(defthm INST-at-stg-INST-stg-LSU
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (INST-in i MT)
                (LSU-stg-p (INST-stg i)))
           (equal (INST-at-stg (INST-stg i) MT) i))
  :hints (("goal" :in-theory (enable LSU-stg-p))))
) ;encapsulate

(defthm INST-at-stg-inst-stg-execute
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (INST-in i MT) (INST-p i)
                (execute-stg-p (INST-stg i)))
           (equal (INST-at-stg (INST-stg i) MT) i))
  :hints (("goal" :in-theory (enable execute-stg-p))))

(defthm INST-at-stg-inst-stg-execute-2
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) stg)
                (execute-stg-p stg))
           (equal (INST-at-stg stg MT) i))
  :hints (("goal" :in-theory (enable execute-stg-p))))
(in-theory (disable INST-at-stg-inst-stg-execute-2))

(defthm INST-at-stg-INST-stg-complete-wbuf0
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (INST-in i MT)
                (equal (INST-stg I) '(complete wbuf0)))
```

```
            (equal (INST-at-stg '(complete wbuf0) MT) i))
  :hints (("goal" :in-theory (e/d (no-stage-conflict
                                   strong-inst-at-stg-theory
                                   no-LSU-stg-conflict)
                                  (no-stage-conflict-p-forward))
                  :use ((:instance no-stage-conflict-p-forward (MA MA)))
                  :restrict ((uniq-inst-at-stg-if-uniq-inst-at-stgs
                              ((stgs '((LSU wbuf0)
                                       (LSU WBUF0 LCH)
                                       (complete wbuf0)
                                       (commit wbuf0)))))))))

(defthm INST-at-stg-INST-stg-complete-wbuf1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (equal (INST-stg I) '(complete wbuf1)))
             (equal (INST-at-stg '(complete wbuf1) MT) i))
  :hints (("goal" :in-theory (e/d (no-stage-conflict
                                   strong-inst-at-stg-theory
                                   no-LSU-stg-conflict)
                                  (no-stage-conflict-p-forward))
                  :use ((:instance no-stage-conflict-p-forward (MA MA)))
                  :restrict ((uniq-inst-at-stg-if-uniq-inst-at-stgs
                              ((stgs '((LSU wbuf1)
                                       (LSU WBUF1 LCH)
                                       (complete wbuf1)
                                       (commit wbuf1)))))))))

(encapsulate nil
(local
(defthm INST-at-stg-INST-stg-commit-wbuf0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (equal (INST-stg I) '(commit wbuf0)))
             (equal (INST-at-stg '(commit wbuf0) MT) i))
  :hints (("goal" :in-theory (e/d (no-stage-conflict
                                   strong-inst-at-stg-theory
                                   no-LSU-stg-conflict)
                                  (no-stage-conflict-p-forward))
                  :use ((:instance no-stage-conflict-p-forward (MA MA)))
                  :restrict ((uniq-inst-at-stg-if-uniq-inst-at-stgs
                              ((stgs '((LSU wbuf0)
                                       (LSU WBUF0 LCH)
                                       (complete wbuf0)
                                       (commit wbuf0)))))))))

(local
(defthm INST-at-stg-INST-stg-commit-wbuf1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (equal (INST-stg I) '(commit wbuf1)))
             (equal (INST-at-stg '(commit wbuf1) MT) i))
  :hints (("goal" :in-theory (e/d (no-stage-conflict
                                   strong-inst-at-stg-theory
                                   no-LSU-stg-conflict)
                                  (no-stage-conflict-p-forward))
                  :use ((:instance no-stage-conflict-p-forward (MA MA)))
                  :restrict ((uniq-inst-at-stg-if-uniq-inst-at-stgs
                              ((stgs '((LSU wbuf1)
```

```
                                    (LSU WBUF1 LCH)
                                    (complete wbuf1)
                                    (commit wbuf1)))))))))))

(defthm INST-at-stg-inst-stg-commit
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (commit-stg-p (INST-stg i)))
             (equal (INST-at-stg (INST-stg i) MT) i))
  :hints (("goal" :in-theory (enable commit-stg-p))))
)

(encapsulate nil
(local
(defthm INST-at-stgs-if-INST-in-help-help
    (implies (and (member-equal i trace)
                  (member-equal (INST-stg i) stgs))
             (not (no-inst-at-stgs-in-trace stgs trace)))))

(local
(defthm INST-at-stgs-if-INST-in-help
    (implies (and (member-equal i trace)
                  (uniq-inst-at-stgs-in-trace stgs trace)
                  (member-equal (INST-stg i) stgs))
             (equal (INST-at-stgs-in-trace stgs trace) i))))

(defthm INST-at-stgs-if-INST-in
    (implies (and (inv MT MA)
                  (INST-p I) (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stgs stgs MT)
                  (member-equal (INST-stg i) stgs))
             (equal (INST-at-stgs stgs MT) i))
  :Hints (("goal" :in-theory (enable INST-in uniq-inst-at-stgs
                                     INST-at-stgs))))
)
(in-theory (disable INST-at-stgs-if-INST-in))
```

D.5.3 MAETT-lemmas2.tex

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MAETT-lemmas2.lisp
; Author  Jun Sawada, University of Texas at Austin
;
;  This book contains various lemmas about the FM9801 and its MAETT
;  abstraction. This book is a continuation of MAETT-lemmas2.lisp.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(in-package "ACL2")

(include-book "MA2-lemmas")
(include-book "invariants-def")
(include-book "MAETT-lemmas1")

(deflabel begin-MAETT-lemmas2)

;  Index
;   Lemmas about relations between MT and MA
;     relations between exception events
```

```
;     (this part occupies 60% of the whole file)
;   Lemmas about micro-architecture satisfying invariants.
;   Lemmas about stages after step-INST, again.
;   Lemmas about no-commit-inst-p and no-dispatched-inst-p
;   Lemmas about MT-non-commit-trace and MT-non-retire-trace.
;   Lemmas about commit again.
;   Lemmas about MT-CMI-p
;   Lemmas about MT-no-jmp-exintr-before

;;;;;;;;;;;;;;;;;Lemmas about MT and MA;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;    Field lemmas
;    Lemmas about ROB and instructions in it.
;    Instruction specific properties about INST fields
;    Relations about INST predicates such as INST-cause-jmp, INST-exintr-now
;       INST-start-specultv?
;    Other lemmas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; IFU field lemmas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deflabel begin-IFU-field-lemmas)
(defthm IFU-valid-if-inst-in
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (IFU-stg-p (INST-stg i)))
             (equal (IFU-valid? (MA-IFU MA)) 1))
  :hints (("Goal" :in-theory (e/d (inst-inv-def
                                   equal-b1p-converter)
                                  (INST-INV-IF-INST-IN))
                  :use (:instance INST-INV-IF-INST-IN))))

(defthm IFU-pc-INST-pc
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (IFU-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (or (not (b1p (INST-modified? i)))
                      (b1p (INST-first-modified? i))))
             (equal (IFU-pc (MA-IFU MA))
                    (INST-pc i)))
  :hints (("Goal" :in-theory (e/d (inst-inv-def
                                   INST-pc
                                   equal-b1p-converter)
                                  (INST-INV-IF-INST-IN))
                  :use (:instance INST-INV-IF-INST-IN))))

(defthm IFU-pc-INST-pc-2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg '(IFU) MT)
                  (not (b1p (inst-specultv? (INST-at-stg '(IFU) MT))))
                  (or (not (b1p (INST-modified? (INST-at-stg '(IFU) MT))))
                      (b1p (INST-first-modified? (INST-at-stg '(IFU) MT)))))
             (equal (IFU-pc (MA-IFU MA))
                    (INST-pc (INST-at-stg '(IFU) MT))))
  :hints (("Goal" :use (:instance IFU-pc-INST-pc
                                  (i (INST-at-stg '(IFU) MT))))))

(defthm IFU-word-INST-word
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
```

```
                (INST-in i MT)
                (IFU-stg-p (INST-stg i))
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (INST-fetch-error-detected-p I)))
           (equal (IFU-word (MA-IFU MA))
                  (INST-word I)))
  :hints (("Goal" :in-theory (e/d (inst-inv-def
                                   equal-b1p-converter)
                                  (INST-INV-IF-INST-IN))
                  :use (:instance INST-INV-IF-INST-IN))))

(defthm IFU-word-INST-word-if-fetch-error
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (IFU-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (INST-fetch-error-detected-p I))
             (equal (IFU-word (MA-IFU MA)) 0))
  :hints (("Goal" :in-theory (e/d (inst-inv-def
                                   equal-b1p-converter)
                                  (INST-INV-IF-INST-IN))
                  :use (:instance INST-INV-IF-INST-IN))))

(defthm IFU-word-INST-word-2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg '(IFU) MT)
                  (not (b1p (inst-specultv? (INST-at-stg '(IFU) MT))))
                  (not (b1p (INST-modified? (INST-at-stg '(IFU) MT)))))
             (equal (IFU-WORD (MA-IFU MA))
                    (IF (INST-fetch-error-detected-p (INST-at-stg '(IFU) MT))
                        0 (INST-word (INST-at-stg '(IFU) MT)))))
  :hints (("Goal" :restrict ((IFU-word-INST-word
                              ((i (INST-at-stg '(IFU) MT))))
                             (IFU-word-INST-word-if-fetch-error
                              ((i (INST-at-stg '(IFU) MT)))))))))

(defthm IFU-excpt-INST-excpt
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (IFU-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (equal (IFU-excpt (MA-IFU MA))
                    (INST-excpt-flags i)))
  :hints (("Goal" :in-theory (e/d (inst-inv-def
                                   equal-b1p-converter)
                                  (INST-INV-IF-INST-IN))
                  :use (:instance INST-INV-IF-INST-IN))))
(deflabel end-IFU-field-lemmas)
(deftheory IFU-field-lemmas
    (set-difference-theories (universal-theory 'begin-IFU-field-lemmas)
                             (universal-theory 'end-IFU-field-lemmas)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; End of IFU field lemmas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
; DQ field lemmas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deflabel begin-DQ-DE0-field-lemmas)

(defthm DQ-DE0-valid-if-inst-in
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(DQ 0)))
             (equal (DE-valid? (DQ-DE0 (MA-DQ MA))) 1))
  :hints (("Goal" :in-theory (e/d (inst-inv-def
                                   equal-b1p-converter)
                                  (INST-INV-IF-INST-IN))
                  :use (:instance INST-INV-IF-INST-IN))))

(defthm DQ-DE1-valid-if-inst-in
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(DQ 1)))
             (equal (DE-valid? (DQ-DE1 (MA-DQ MA))) 1))
  :hints (("Goal" :in-theory (e/d (inst-inv-def
                                   equal-b1p-converter)
                                  (INST-INV-IF-INST-IN))
                  :use (:instance INST-INV-IF-INST-IN))))

(defthm DQ-DE2-valid-if-inst-in
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(DQ 2)))
             (equal (DE-valid? (DQ-DE2 (MA-DQ MA))) 1))
  :hints (("Goal" :in-theory (e/d (inst-inv-def
                                   equal-b1p-converter)
                                  (INST-INV-IF-INST-IN))
                  :use (:instance INST-INV-IF-INST-IN))))

(defthm DQ-DE3-valid-if-inst-in
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(DQ 3)))
             (equal (DE-valid? (DQ-DE3 (MA-DQ MA))) 1))
  :hints (("Goal" :in-theory (e/d (inst-inv-def
                                   equal-b1p-converter)
                                  (INST-INV-IF-INST-IN))
                  :use (:instance INST-INV-IF-INST-IN))))

(defthm uniq-inst-at-stg-DQ-MT-DQ-len-minus-1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (MT-DQ-len MT) 0)))
             (uniq-inst-at-stg (list 'DQ (1- (MT-DQ-len MT))) MT))
  :hints (("Goal" :cases ((equal (MT-DQ-len MT) 1) (equal (MT-DQ-len MT) 2)
                          (equal (MT-DQ-len MT) 3) (equal (MT-DQ-len MT) 4)))
          (when-found (uniq-inst-at-stg '(DQ 0) MT)
                      (:cases ((b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))))
          (when-found (uniq-inst-at-stg '(DQ 1) MT)
                      (:cases ((b1p (DE-valid? (DQ-DE1 (MA-DQ MA)))))))
          (when-found (uniq-inst-at-stg '(DQ 2) MT)
                      (:cases ((b1p (DE-valid? (DQ-DE2 (MA-DQ MA)))))))
          (when-found (uniq-inst-at-stg '(DQ 3) MT)
```

```
                    (:cases ((b1p (DE-valid? (DQ-DE3 (MA-DQ MA)))))))))

(defthm DQ-DE0-cntlv-=-inst-cntlv-1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (equal (INST-stg i) '(DQ 0))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i)))
             (equal (DE-cntlv (DQ-DE0 (MA-DQ MA)))
                    (INST-cntlv i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def
                                   INST-excpt-detected-p
                                   equal-b1p-converter)
                                  (INST-INV-IF-INST-IN))
                  :use (:instance INST-INV-IF-INST-IN))))

(defthm DQ-DE0-cntlv-=-inst-cntlv-2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-INST-at-stg '(DQ 0) MT)
                  (not (b1p (inst-specultv? (INST-at-stg '(DQ 0) MT))))
                  (not (b1p (INST-modified? (INST-at-stg '(DQ 0) MT))))
                  (not (INST-fetch-error-detected-p (INST-at-stg '(DQ 0) MT))))
             (equal (DE-cntlv (DQ-DE0 (MA-DQ MA)))
                    (INST-cntlv (INST-at-stg '(DQ 0) MT))))
  :hints (("goal" :use (:instance DQ-DE0-cntlv-=-inst-cntlv-1
                                  (i (INST-at-stg '(DQ 0) MT))))))

(defthm DQ-DE0-cntlv-=-inst-cntlv-1-if-fetch-error
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (equal (INST-stg i) '(DQ 0))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (INST-fetch-error-detected-p i))
             (equal (DE-cntlv (DQ-DE0 (MA-DQ MA)))
                    (decode 0 (INST-br-predict? i))))
  :hints (("goal" :in-theory (e/d (inst-inv-def
                                   INST-excpt-detected-p
                                   equal-b1p-converter)
                                  (INST-INV-IF-INST-IN))
                  :use (:instance INST-INV-IF-INST-IN))))

(defthm DQ-DE0-PC-=-inst-pc
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (equal (INST-stg i) '(DQ 0))
                  (not (b1p (inst-specultv? i)))
                  (or (not (b1p (INST-modified? i)))
                      (b1p (INST-first-modified? i))))
             (equal (DE-pc (DQ-DE0 (MA-DQ MA)))
                    (INST-pc i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def
                                   INST-excpt-detected-p
                                   INST-pc
                                   equal-b1p-converter)
                                  (INST-INV-IF-INST-IN))
                  :use (:instance INST-INV-IF-INST-IN))))

(defthm DQ-DE0-PC-=-inst-pc-2
    (implies (and (inv MT MA)
```

```
                (MAETT-p MT) (MA-state-p MA)
                (uniq-INST-at-stg '(DQ 0) MT)
                (not (b1p (inst-specultv? (INST-at-stg '(DQ 0) MT))))
                (or (not (b1p (INST-modified? (INST-at-stg '(DQ 0) MT))))
                    (b1p (INST-first-modified? (INST-at-stg '(DQ 0) MT)))))
           (equal (DE-PC (DQ-DE0 (MA-DQ MA)))
                  (INST-pc (INST-at-stg '(DQ 0) MT))))
  :hints (("goal" :use (:instance DQ-DE0-PC-=-inst-pc
                                  (i (INST-at-stg '(DQ 0) MT))))))

(defthm DQ-DE0-RC-=-inst-rc
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (equal (INST-stg i) '(DQ 0))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i)))
             (equal (DE-RC (DQ-DE0 (MA-DQ MA)))
                    (INST-rc i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def
                                   INST-excpt-detected-p
                                   INST-rc
                                   equal-b1p-converter)
                                  (INST-INV-IF-INST-IN))
                  :use (:instance INST-INV-IF-INST-IN))))

(defthm DQ-DE0-RC-=-inst-rc-2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-INST-at-stg '(DQ 0) MT)
                  (not (b1p (inst-specultv? (INST-at-stg '(DQ 0) MT))))
                  (not (b1p (INST-modified? (INST-at-stg '(DQ 0) MT))))
                  (not (INST-fetch-error-detected-p (INST-at-stg '(DQ 0) MT))))
             (equal (DE-RC (DQ-DE0 (MA-DQ MA)))
                    (INST-rc (INST-at-stg '(DQ 0) MT))))
  :hints (("goal" :use (:instance DQ-DE0-RC-=-inst-rc
                                  (i (INST-at-stg '(DQ 0) MT))))))

(defthm DQ-DE0-RA-=-inst-ra
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (equal (INST-stg i) '(DQ 0))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i)))
             (equal (DE-RA (DQ-DE0 (MA-DQ MA)))
                    (INST-ra i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def
                                   INST-excpt-detected-p
                                   INST-ra
                                   equal-b1p-converter)
                                  (INST-INV-IF-INST-IN))
                  :use (:instance INST-INV-IF-INST-IN))))

(defthm DQ-DE0-RA-=-inst-ra-2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-INST-at-stg '(DQ 0) MT)
                  (not (b1p (inst-specultv? (INST-at-stg '(DQ 0) MT))))
                  (not (b1p (INST-modified? (INST-at-stg '(DQ 0) MT))))
                  (not (INST-fetch-error-detected-p (INST-at-stg '(DQ 0) MT))))
             (equal (DE-RA (DQ-DE0 (MA-DQ MA)))
```

```
                  (INST-ra (INST-at-stg '(DQ 0) MT))))
  :hints (("goal" :use (:instance DQ-DE0-RA-=-inst-ra
                                  (i (INST-at-stg '(DQ 0) MT))))))

(defthm DQ-DE0-RB-=-inst-rb
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (equal (INST-stg i) '(DQ 0))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i)))
             (equal (DE-RB (DQ-DE0 (MA-DQ MA)))
                    (INST-rb i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def
                                   INST-excpt-detected-p
                                   INST-rb
                                   equal-b1p-converter)
                                  (INST-INV-IF-INST-IN))
                  :use (:instance INST-INV-IF-INST-IN))))

(defthm DQ-DE0-RB-=-inst-rb-2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-INST-at-stg '(DQ 0) MT)
                  (not (b1p (inst-specultv? (INST-at-stg '(DQ 0) MT))))
                  (not (b1p (INST-modified? (INST-at-stg '(DQ 0) MT))))
                  (not (INST-fetch-error-detected-p (INST-at-stg '(DQ 0) MT))))
             (equal (DE-RB (DQ-DE0 (MA-DQ MA)))
                    (INST-rb (INST-at-stg '(DQ 0) MT))))
  :hints (("goal" :use (:instance DQ-DE0-RB-=-inst-rb
                                  (i (INST-at-stg '(DQ 0) MT))))))

(defthm DQ-DE0-IM-=-inst-im
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (equal (INST-stg i) '(DQ 0))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i)))
             (equal (DE-IM (DQ-DE0 (MA-DQ MA)))
                    (INST-im i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def
                                   INST-im
                                   INST-excpt-detected-p
                                   equal-b1p-converter)
                                  (INST-INV-IF-INST-IN))
                  :use (:instance INST-INV-IF-INST-IN))))

(defthm DQ-DE0-im-=-inst-im-2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-INST-at-stg '(DQ 0) MT)
                  (not (b1p (inst-specultv? (INST-at-stg '(DQ 0) MT))))
                  (not (b1p (INST-modified? (INST-at-stg '(DQ 0) MT))))
                  (not (INST-fetch-error-detected-p (INST-at-stg '(DQ 0) MT))))
             (equal (DE-IM (DQ-DE0 (MA-DQ MA)))
                    (INST-im (INST-at-stg '(DQ 0) MT))))
  :hints (("goal" :use (:instance DQ-DE0-im-=-inst-im
                                  (i (INST-at-stg '(DQ 0) MT))))))

(defthm DQ-DE0-br-target-=-inst-br-target
    (implies (and (inv MT MA)
```

```
                (MAETT-p MT) (MA-state-p MA)
                (INST-in i MT) (equal (INST-stg i) '(DQ 0))
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (INST-fetch-error-detected-p i)))
           (equal (DE-BR-target (DQ-DE0 (MA-DQ MA)))
                  (INST-br-target i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def
                                   INST-excpt-detected-p
                                   equal-b1p-converter)
                                  (INST-INV-IF-INST-IN))
                  :use (:instance INST-INV-IF-INST-IN))))

(defthm DQ-DE0-br-target-=-inst-br-target-2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-INST-at-stg '(DQ 0) MT)
                  (not (b1p (inst-specultv? (INST-at-stg '(DQ 0) MT))))
                  (not (b1p (INST-modified? (INST-at-stg '(DQ 0) MT))))
                  (not (INST-fetch-error-detected-p (INST-at-stg '(DQ 0) MT))))
             (equal (DE-BR-target (DQ-DE0 (MA-DQ MA)))
                    (INST-br-target (INST-at-stg '(DQ 0) MT))))
  :hints (("goal" :use (:instance DQ-DE0-BR-target-=-inst-br-target
                                  (i (INST-at-stg '(DQ 0) MT))))))

(defthm DQ-DE0-excpt-=-INST-excpt-flags
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (equal (INST-stg i) '(DQ 0))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (equal (DE-excpt (DQ-DE0 (MA-DQ MA)))
                    (INST-excpt-flags i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def
                                   INST-excpt-detected-p
                                   equal-b1p-converter)
                                  (INST-INV-IF-INST-IN))
                  :use (:instance INST-INV-IF-INST-IN))))

(defthm DQ-DE0-excpt-=-inst-excpt-flags-2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-INST-at-stg '(DQ 0) MT)
                  (not (b1p (inst-specultv? (INST-at-stg '(DQ 0) MT))))
                  (not (b1p (INST-modified? (INST-at-stg '(DQ 0) MT)))))
             (equal (DE-excpt (DQ-DE0 (MA-DQ MA)))
                    (INST-excpt-flags (INST-at-stg '(DQ 0) MT))))
  :hints (("goal" :use (:instance DQ-DE0-excpt-=-INST-excpt-flags
                                  (i (INST-at-stg '(DQ 0) MT))))))
(deflabel end-DQ-DE0-field-lemmas)
(deftheory DQ-DE0-field-lemmas
    (set-difference-theories (universal-theory 'end-DQ-DE0-field-lemmas)
                             (universal-theory 'begin-DQ-DE0-field-lemmas)))

;;;;;;;;;;;;End of DQ field lemmas;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;Field of Dispatch Output;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; The cntlv of a dispatched instruction is represented with INST-cntlv
; of an instruction i at (DQ 0).
(defthm dispatch-cntlv-INST-cntlv
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
```

```
                (equal (INST-stg i) '(DQ 0))
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (INST-fetch-error-detected-p i)))
           (equal (dispatch-cntlv MA) (INST-cntlv i)))
  :hints (("goal" :in-theory (enable dispatch-cntlv)
                  :cases ((INST-fetch-error-detected-p i)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(deflabel begin-INST-types-at-execution-units)
; Only a certain type of instruction can be in an execution unit in a
; reachable state.  For instance, only an integer unit instruction can
; be in the Integer Unit, but not a multiply instruction or branch
; instruction.
(defthm INST-IU-if-IU-stg-p
    (implies (and (inv MT MA)
                  (IU-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i))
             (equal (INST-IU? i) 1))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter
                                                IU-stg-p)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm INST-MU-if-MU-stg-p
    (implies (and (inv MT MA)
                  (MU-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i))
             (equal (INST-MU? i) 1))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter
                                                MU-stg-p)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm INST-BU-if-BU-stg-p
    (implies (and (inv MT MA)
                  (BU-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i))
             (equal (INST-BU? i) 1))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter
                                                BU-stg-p)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm INST-LSU-if-LSU-stg-p
    (implies (and (inv MT MA)
                  (LSU-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i))
             (equal (INST-LSU? i) 1))
```

```
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter
                                                LSU-stg-p)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))
(deflabel end-INST-types-at-execution-units)

(deftheory INST-types-at-execution-units
    (set-difference-theories
     (universal-theory 'end-INST-types-at-execution-units)
     (universal-theory 'begin-INST-types-at-execution-units)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  IU RS field lemmas
(deflabel begin-IU-RS-field-lemmas)
(defthm IU-RS0-valid-if-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(IU RS0))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (RS-valid? (IU-RS0 (MA-IU MA))) 1))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm IU-RS1-valid-if-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(IU RS1))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (RS-valid? (IU-RS1 (MA-IU MA))) 1))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm uniq-inst-at-IU-RS0-if-valid
    (implies (and (inv MT MA)
                  (b1p (RS-valid? (IU-RS0 (MA-IU MA))))
                  (MA-state-p MA) (MAETT-p MT))
             (uniq-INST-at-stg '(IU RS0) MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-IU-stg-conflict))))

(defthm uniq-inst-at-IU-RS1-if-valid
    (implies (and (inv MT MA)
                  (b1p (RS-valid? (IU-RS1 (MA-IU MA))))
                  (MA-state-p MA) (MAETT-p MT))
             (uniq-INST-at-stg '(IU RS1) MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-IU-stg-conflict))))

(defthm no-inst-at-IU-RS0
    (implies (and (inv MT MA)
                  (not (b1p (RS-valid? (IU-RS0 (MA-IU MA)))))
                  (MA-state-p MA) (MAETT-p MT))
             (no-INST-at-stg '(IU RS0) MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-IU-stg-conflict))))

(defthm no-inst-at-IU-RS1
    (implies (and (inv MT MA)
```

```
                (not (b1p (RS-valid? (IU-RS1 (MA-IU MA)))))
                (MA-state-p MA) (MAETT-p MT))
           (no-INST-at-stg '(IU RS1) MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-IU-stg-conflict))))

(defthm not-INST-excpt-detected-p-if-IU-stg-p
    (implies (and (inv MT MA)
                  (IU-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA))
             (not (INST-excpt-detected-p i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter
                                                IU-stg-p)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm IU-RS0-op-=-inst-IU-op
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(IU RS0))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-op (IU-RS0 (MA-IU MA)))
                    (INST-IU-op? i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm IU-RS0-op-=-inst-IU-op-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(IU RS0) MT)
                  (not (b1p (inst-specultv?
                             (inst-at-stg '(IU RS0) MT))))
                  (not (b1p (INST-modified?
                             (inst-at-stg '(IU RS0) MT))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-op (IU-RS0 (MA-IU MA)))
                    (INST-IU-op? (inst-at-stg '(IU RS0) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (inst-at-stg '(IU RS0) MT))))))

(defthm IU-RS1-op-=-inst-IU-op
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(IU RS1))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-op (IU-RS1 (MA-IU MA)))
                    (INST-IU-op? i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm IU-RS1-op-=-inst-IU-op-2
    (implies (and (inv MT MA)
```

```
                (uniq-inst-at-stg '(IU RS1) MT)
                (not (b1p (inst-specultv?
                           (inst-at-stg '(IU RS1) MT))))
                (not (b1p (INST-modified?
                           (inst-at-stg '(IU RS1) MT))))
                (MAETT-p MT) (MA-state-p MA))
           (equal (RS-op (IU-RS1 (MA-IU MA)))
                  (INST-IU-op? (inst-at-stg '(IU RS1) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (inst-at-stg '(IU RS1) MT))))))

(defthm IU-RS0-tag-=-inst-tag
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(IU RS0))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-tag (IU-RS0 (MA-IU MA)))
                    (INST-tag i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm IU-RS0-tag-=-inst-tag-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(IU RS0) MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-tag (IU-RS0 (MA-IU MA)))
                    (INST-tag (INST-at-stg '(IU RS0) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(IU RS0) MT))))))

(defthm IU-RS-tag-=-inst-tag
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(IU RS1))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-tag (IU-RS1 (MA-IU MA)))
                    (INST-tag i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm IU-RS-tag-=-inst-tag-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(IU RS1) MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-tag (IU-RS1 (MA-IU MA)))
                    (INST-tag (INST-at-stg '(IU RS1) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(IU RS1) MT))))))

(defthm IU-RS0-val1-=-INST-src-val1
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(IU RS0))
                  (not (b1p (inst-specultv? i)))
```

```
                (not (b1p (INST-modified? i)))
                (b1p (RS-ready1? (IU-RS0 (MA-IU MA))))
                (MAETT-p MT) (MA-state-p MA))
           (equal (RS-val1 (IU-RS0 (MA-IU MA)))
                  (INST-src-val1 i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm IU-RS0-val1-=-INST-src-val1-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(IU RS0) MT)
                  (not (b1p (inst-specultv?
                             (inst-at-stg '(IU RS0) MT))))
                  (not (b1p (INST-modified?
                             (inst-at-stg '(IU RS0) MT))))
                  (b1p (RS-ready1? (IU-RS0 (MA-IU MA))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-val1 (IU-RS0 (MA-IU MA)))
                    (INST-src-val1
                     (inst-at-stg '(IU RS0) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (inst-at-stg '(IU RS0) MT))))))

(defthm IU-RS1-val1-=-INST-src-val1
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(IU RS1))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (b1p (RS-ready1? (IU-RS1 (MA-IU MA))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-val1 (IU-RS1 (MA-IU MA)))
                    (INST-src-val1 i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm IU-RS1-val1-=-INST-src-val1-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(IU RS1) MT)
                  (not (b1p (inst-specultv?
                             (INST-at-stg '(IU RS1) MT))))
                  (not (b1p (INST-modified?
                             (INST-at-stg '(IU RS1) MT))))
                  (b1p (RS-ready1? (IU-RS1 (MA-IU MA))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-val1 (IU-RS1 (MA-IU MA)))
                    (INST-src-val1 (INST-at-stg '(IU RS1) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(IU RS1) MT))))))

(defthm IU-RS0-val2-=-INST-src-val2
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(IU RS0))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
```

```
                (b1p (RS-ready2? (IU-RS0 (MA-IU MA))))
                (not (b1p (INST-IU-op? i)))
                (MAETT-p MT) (MA-state-p MA))
           (equal (RS-val2 (IU-RS0 (MA-IU MA)))
                  (INST-src-val2 i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm IU-RS0-val2-=-INST-src-val2-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(IU RS0) MT)
                  (not (b1p (inst-specultv?
                             (INST-at-stg '(IU RS0) MT))))
                  (not (b1p (INST-modified?
                             (INST-at-stg '(IU RS0) MT))))
                  (b1p (RS-ready2? (IU-RS0 (MA-IU MA))))
                  (not (b1p (INST-IU-op?
                             (INST-at-stg '(IU RS0) MT))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-val2 (IU-RS0 (MA-IU MA)))
                    (INST-src-val2 (INST-at-stg '(IU RS0) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(IU RS0) MT))))))

(defthm IU-RS1-val2-=-INST-src-val2
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(IU RS1))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (b1p (RS-ready2? (IU-RS1 (MA-IU MA))))
                  (not (b1p (INST-IU-op? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-val2 (IU-RS1 (MA-IU MA)))
                    (INST-src-val2 i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm IU-RS1-val2-=-INST-src-val2-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(IU RS1) MT)
                  (not (b1p (inst-specultv?
                             (inst-at-stg  '(IU RS1) MT))))
                  (not (b1p (INST-modified?
                             (inst-at-stg  '(IU RS1) MT))))
                  (b1p (RS-ready2? (IU-RS1 (MA-IU MA))))
                  (not (b1p (INST-IU-op?
                             (inst-at-stg  '(IU RS1) MT))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-val2 (IU-RS1 (MA-IU MA)))
                    (INST-src-val2 (inst-at-stg  '(IU RS1) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (inst-at-stg  '(IU RS1) MT))))))

(deflabel end-IU-RS-field-lemmas)
(deftheory IU-RS-field-lemmas
```

```
    (set-difference-theories
     (universal-theory 'end-IU-RS-field-lemmas)
     (universal-theory 'begin-IU-RS-field-lemmas)))

(deftheory IU-RS-field-INST-at-lemmas
    '(IU-RS0-op-=-inst-IU-op-2 IU-RS1-op-=-inst-IU-op-2
      IU-RS0-tag-=-inst-tag-2 IU-RS-tag-=-inst-tag-2
      IU-RS0-val1-=-INST-src-val1-2 IU-RS1-val1-=-INST-src-val1-2
      IU-RS0-val2-=-INST-src-val2-2 IU-RS1-val2-=-INST-src-val2-2))
(in-theory (disable IU-RS-field-INST-at-lemmas))

;;;;;;;;;;;;;;;;;;;BU field lemmas;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(deflabel begin-BU-RS-field-lemmas)
(defthm BU-RS0-valid-if-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(BU RS0))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (BU-RS-valid? (BU-RS0 (MA-BU MA))) 1))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm BU-RS1-valid-if-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(BU RS1))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (BU-RS-valid? (BU-RS1 (MA-BU MA))) 1))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm uniq-inst-at-BU-RS0-if-valid
    (implies (and (inv MT MA)
                  (b1p (BU-RS-valid? (BU-RS0 (MA-BU MA))))
                  (MA-state-p MA) (MAETT-p MT))
             (uniq-INST-at-stg '(BU RS0) MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-BU-stg-conflict))))

(defthm uniq-inst-at-BU-RS1-if-valid
    (implies (and (inv MT MA)
                  (b1p (BU-RS-valid? (BU-RS1 (MA-BU MA))))
                  (MA-state-p MA) (MAETT-p MT))
             (uniq-INST-at-stg '(BU RS1) MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-BU-stg-conflict))))

(defthm no-inst-at-BU-RS0
    (implies (and (inv MT MA)
                  (not (b1p (BU-RS-valid? (BU-RS0 (MA-BU MA)))))
                  (MA-state-p MA) (MAETT-p MT))
             (no-INST-at-stg '(BU RS0) MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-BU-stg-conflict))))

(defthm no-inst-at-BU-RS1
    (implies (and (inv MT MA)
```

```
                (not (b1p (BU-RS-valid? (BU-RS1 (MA-BU MA)))))
                (MA-state-p MA) (MAETT-p MT))
           (no-INST-at-stg '(BU RS1) MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-BU-stg-conflict))))

(defthm BU-RS0-tag-=-inst-tag
    (implies (and (inv MT MA)
                (INST-in i MT)
                (equal (INST-stg i) '(BU RS0))
                (MAETT-p MT) (MA-state-p MA))
           (equal (BU-RS-tag (BU-RS0 (MA-BU MA)))
                  (INST-tag i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                :use (:instance INST-inv-if-INST-in))))

(defthm BU-RS0-tag-=-inst-tag-2
    (implies (and (inv MT MA)
                (uniq-inst-at-stg '(BU RS0) MT)
                (MAETT-p MT) (MA-state-p MA))
           (equal (BU-RS-tag (BU-RS0 (MA-BU MA)))
                  (INST-tag (INST-at-stg '(BU RS0) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                :use (:instance INST-inv-if-INST-in
                                (i (INST-at-stg '(BU RS0) MT))))))

(defthm BU-RS-tag-=-inst-tag
    (implies (and (inv MT MA)
                (INST-in i MT)
                (equal (INST-stg i) '(BU RS1))
                (MAETT-p MT) (MA-state-p MA))
           (equal (BU-RS-tag (BU-RS1 (MA-BU MA)))
                  (INST-tag i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                :use (:instance INST-inv-if-INST-in))))

(defthm BU-RS-tag-=-inst-tag-2
    (implies (and (inv MT MA)
                (uniq-inst-at-stg '(BU RS1) MT)
                (MAETT-p MT) (MA-state-p MA))
           (equal (BU-RS-tag (BU-RS1 (MA-BU MA)))
                  (INST-tag (INST-at-stg '(BU RS1) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                :use (:instance INST-inv-if-INST-in
                                (i (INST-at-stg '(BU RS1) MT))))))

(defthm BU-RS0-val-=-INST-src-val1
    (implies (and (inv MT MA)
                (INST-in i MT)
                (equal (INST-stg i) '(BU RS0))
                (b1p (BU-RS-ready? (BU-RS0 (MA-BU MA))))
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (MAETT-p MT) (MA-state-p MA))
           (equal (BU-RS-val (BU-RS0 (MA-BU MA)))
                  (INST-src-val3 i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
```

```
                :use (:instance INST-inv-if-INST-in))))

(defthm BU-RS0-val-=-INST-src-val1-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(BU RS0) MT)
                  (b1p (BU-RS-ready? (BU-RS0 (MA-BU MA))))
                  (not (b1p (inst-specultv?
                             (INST-at-stg '(BU RS0) MT))))
                  (not (b1p (INST-modified?
                             (INST-at-stg '(BU RS0) MT))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (BU-RS-val (BU-RS0 (MA-BU MA)))
                    (INST-src-val3 (INST-at-stg '(BU RS0) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(BU RS0) MT))))))

(defthm BU-RS1-val-=-INST-src-val1
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(BU RS1))
                  (b1p (BU-RS-ready? (BU-RS1 (MA-BU MA))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (BU-RS-val (BU-RS1 (MA-BU MA)))
                    (INST-src-val3 i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm BU-RS1-val-=-INST-src-val1-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(BU RS1) MT)
                  (b1p (BU-RS-ready? (BU-RS1 (MA-BU MA))))
                  (not (b1p (inst-specultv?
                             (INST-at-stg '(BU RS1) MT))))
                  (not (b1p (INST-modified?
                             (INST-at-stg '(BU RS1) MT))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (BU-RS-val (BU-RS1 (MA-BU MA)))
                    (INST-src-val3 (INST-at-stg '(BU RS1) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(BU RS1) MT))))))

(deflabel end-BU-RS-field-lemmas)
(deftheory BU-RS-field-lemmas
    (set-difference-theories
     (universal-theory 'end-BU-RS-field-lemmas)
     (universal-theory 'begin-BU-RS-field-lemmas)))
(deftheory BU-RS-field-INST-at-lemmas
    '(BU-RS0-tag-=-inst-tag-2 BU-RS-tag-=-inst-tag-2
      BU-RS1-val-=-INST-src-val1-2 BU-RS0-val-=-INST-src-val1-2))
(in-theory (disable BU-RS-field-INST-at-lemmas))

;;;;;;;;;;;;;;;;;;;;MU field lemmas;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deflabel begin-MU-field-lemmas)
(deflabel begin-MU-RS-field-lemmas)
```

```
(defthm MU-RS0-valid-if-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(MU RS0))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (RS-valid? (MU-RS0 (MA-MU MA))) 1))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm MU-RS1-valid-if-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(MU RS1))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (RS-valid? (MU-RS1 (MA-MU MA))) 1))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm uniq-inst-at-MU-RS0-if-valid
    (implies (and (inv MT MA)
                  (b1p (RS-valid? (MU-rs0 (MA-MU MA))))
                  (MA-state-p MA) (MAETT-p MT))
             (uniq-INST-at-stg '(MU RS0) MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-MU-stg-conflict))))

(defthm uniq-inst-at-MU-RS1-if-valid
    (implies (and (inv MT MA)
                  (b1p (RS-valid? (MU-rs1 (MA-MU MA))))
                  (MA-state-p MA) (MAETT-p MT))
             (uniq-INST-at-stg '(MU RS1) MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-MU-stg-conflict))))

(defthm no-inst-at-MU-RS0
    (implies (and (inv MT MA)
                  (not (b1p (RS-valid? (MU-rs0 (MA-MU MA)))))
                  (MA-state-p MA) (MAETT-p MT))
             (no-INST-at-stg '(MU RS0) MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-MU-stg-conflict))))

(defthm no-inst-at-MU-RS1
    (implies (and (inv MT MA)
                  (not (b1p (RS-valid? (MU-rs1 (MA-MU MA)))))
                  (MA-state-p MA) (MAETT-p MT))
             (no-INST-at-stg '(MU RS1) MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-MU-stg-conflict))))

(defthm MU-RS0-tag-=-inst-tag
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(MU RS0))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-tag (MU-RS0 (MA-MU MA)))
                    (INST-tag i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
```

```
                                (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm MU-RS0-tag-=-inst-tag-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(MU RS0) MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-tag (MU-RS0 (MA-MU MA)))
                    (INST-tag (INST-at-stg '(MU RS0) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(MU RS0) MT))))))

(defthm MU-RS-tag-=-inst-tag
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(MU RS1))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-tag (MU-RS1 (MA-MU MA)))
                    (INST-tag i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm MU-RS-tag-=-inst-tag-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(MU RS1) MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-tag (MU-RS1 (MA-MU MA)))
                    (INST-tag (INST-at-stg '(MU RS1) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(MU RS1) MT))))))

(defthm MU-RS0-val1-=-INST-src-val1
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(MU RS0))
                  (b1p (RS-ready1? (MU-RS0 (MA-MU MA))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-val1 (MU-RS0 (MA-MU MA))) (INST-src-val1 i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm MU-RS0-val1-=-INST-src-val1-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(MU RS0) MT)
                  (b1p (RS-ready1? (MU-RS0 (MA-MU MA))))
                  (not (b1p (inst-specultv?
                             (INST-at-stg '(MU RS0) MT))))
                  (not (b1p (INST-modified?
                             (INST-at-stg '(MU RS0) MT))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-val1 (MU-RS0 (MA-MU MA)))
                    (INST-src-val1 (INST-at-stg '(MU RS0) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
```

```
                :use (:instance INST-inv-if-INST-in
                                (i (INST-at-stg '(MU RS0) MT))))))

(defthm MU-RS1-val1-=-INST-src-val1
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(MU RS1))
                  (b1p (RS-ready1? (MU-RS1 (MA-MU MA))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-val1 (MU-RS1 (MA-MU MA))) (INST-src-val1 i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm MU-RS1-val1-=-INST-src-val1-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(MU RS1) MT)
                  (b1p (RS-ready1? (MU-RS1 (MA-MU MA))))
                  (not (b1p (inst-specultv?
                             (INST-at-stg '(MU RS1) MT))))
                  (not (b1p (INST-modified?
                             (INST-at-stg '(MU RS1) MT))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-val1 (MU-RS1 (MA-MU MA)))
                    (INST-src-val1 (INST-at-stg '(MU RS1) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(MU RS1) MT))))))

(defthm MU-RS0-val2-=-INST-src-val2
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(MU RS0))
                  (b1p (RS-ready2? (MU-RS0 (MA-MU MA))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-val2 (MU-RS0 (MA-MU MA))) (INST-src-val2 i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm MU-RS0-val2-=-INST-src-val2-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(MU RS0) MT)
                  (b1p (RS-ready2? (MU-RS0 (MA-MU MA))))
                  (not (b1p (inst-specultv?
                             (INST-at-stg '(MU RS0) MT))))
                  (not (b1p (INST-modified?
                             (INST-at-stg '(MU RS0) MT))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-val2 (MU-RS0 (MA-MU MA)))
                    (INST-src-val2 (INST-at-stg '(MU RS0) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(MU RS0) MT))))))

(defthm MU-RS1-val2-=-INST-src-val2
```

```
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(MU RS1))
                  (b1p (RS-ready2? (MU-RS1 (MA-MU MA))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-val2 (MU-RS1 (MA-MU MA))) (INST-src-val2 i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm MU-RS1-val2-=-INST-src-val2-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(MU RS1) MT)
                  (b1p (RS-ready2? (MU-RS1 (MA-MU MA))))
                  (not (b1p (inst-specultv?
                             (INST-at-stg '(MU RS1) MT))))
                  (not (b1p (INST-modified?
                             (INST-at-stg '(MU RS1) MT))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (RS-val2 (MU-RS1 (MA-MU MA)))
                    (INST-src-val2 (INST-at-stg '(MU RS1) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(MU RS1) MT))))))

(deflabel end-MU-RS-field-lemmas)
(deftheory MU-RS-field-inst-at-lemmas
    '(MU-RS0-tag-=-inst-tag-2 MU-RS-tag-=-inst-tag-2
      MU-RS0-val1-=-INST-src-val1-2 MU-RS1-val1-=-INST-src-val1-2
      MU-RS0-val2-=-INST-src-val2-2 MU-RS1-val2-=-INST-src-val2-2))
(in-theory (disable MU-RS-field-inst-at-lemmas))

(deflabel begin-MU-latch-field-lemmas)

(defthm MU-lch1-valid-if-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(MU lch1))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (MU-latch1-valid? (MU-lch1 (MA-MU MA))) 1))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm uniq-inst-at-MU-lch1-if-valid
    (implies (and (inv MT MA)
                  (b1p (MU-latch1-valid? (MU-lch1 (MA-MU MA))))
                  (MA-state-p MA) (MAETT-p MT))
             (uniq-INST-at-stg '(MU lch1) MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-MU-stg-conflict))))
(defthm no-inst-at-MU-lch1
    (implies (and (inv MT MA)
                  (not (b1p (MU-latch1-valid? (MU-lch1 (MA-MU MA)))))
                  (MA-state-p MA) (MAETT-p MT))
             (no-INST-at-stg '(MU lch1) MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-MU-stg-conflict))))
```

```
(defthm MU-lch1-tag-=-inst-tag
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(MU lch1))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (MU-latch1-tag (MU-lch1 (MA-MU MA)))
                    (INST-tag i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm MU-lch1-tag-=-inst-tag-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(MU lch1) MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (MU-latch1-tag (MU-lch1 (MA-MU MA)))
                    (INST-tag (INST-at-stg '(MU lch1) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(MU lch1) MT))))))

(defthm MU-lch1-data-=-ML1-output
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(MU lch1))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (MU-latch1-data (MU-lch1 (MA-MU MA)))
                    (ML1-output (INST-src-val1 i) (INST-src-val2 i))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm MU-lch1-data-=-ML1-output-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(MU lch1) MT)
                  (not (b1p (inst-specultv?
                             (INST-at-stg '(MU lch1) MT))))
                  (not (b1p (INST-modified?
                             (INST-at-stg '(MU lch1) MT))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (MU-latch1-data (MU-lch1 (MA-MU MA)))
                    (ML1-output (INST-src-val1 (INST-at-stg '(MU lch1) MT))
                                (INST-src-val2 (INST-at-stg '(MU lch1) MT)))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(MU lch1) MT))))))
(deftheory MU-lch1-field-INST-at-lemmas
    '(MU-lch1-tag-=-inst-tag-2 MU-lch1-data-=-ML1-output-2))
(in-theory (disable MU-lch1-field-INST-at-lemmas))

(defthm MU-lch2-valid-if-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(MU lch2))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (MU-latch2-valid? (MU-lch2 (MA-MU MA))) 1))
```

```
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm uniq-inst-at-MU-lch2-if-valid
    (implies (and (inv MT MA)
                  (b1p (MU-latch2-valid? (MU-lch2 (MA-MU MA))))
                  (MA-state-p MA) (MAETT-p MT))
             (uniq-INST-at-stg '(MU lch2) MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-MU-stg-conflict))))

(defthm no-inst-at-MU-lch2
    (implies (and (inv MT MA)
                  (not (b1p (MU-latch2-valid? (MU-lch2 (MA-MU MA)))))
                  (MA-state-p MA) (MAETT-p MT))
             (no-INST-at-stg '(MU lch2) MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-MU-stg-conflict))))
(defthm MU-lch2-tag-=-inst-tag
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(MU lch2))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (MU-latch2-tag (MU-lch2 (MA-MU MA)))
                    (INST-tag i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm MU-lch2-tag-=-inst-tag-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(MU lch2) MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (MU-latch2-tag (MU-lch2 (MA-MU MA)))
                    (INST-tag (INST-at-stg '(MU lch2) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(MU lch2) MT))))))

(defthm MU-lch2-data-=-ML2-output-ML2-output
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(MU lch2))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (MU-latch2-data (MU-lch2 (MA-MU MA)))
                    (ML2-output (ML1-output (INST-src-val1 i)
                                            (INST-src-val2 i)))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm MU-lch2-data-=-ML2-output-ML2-output-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(MU lch2) MT)
                  (not (b1p (inst-specultv?
                             (INST-at-stg '(MU lch2) MT))))
                  (not (b1p (INST-modified?
                             (INST-at-stg '(MU lch2) MT))))
```

```
                (MAETT-p MT) (MA-state-p MA))
           (equal (MU-latch2-data (MU-lch2 (MA-MU MA)))
                  (ML2-output
                   (ML1-output (INST-src-val1
                                (INST-at-stg '(MU lch2) MT))
                               (INST-src-val2
                                (INST-at-stg '(MU lch2) MT))))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(MU lch2) MT))))))
(deflabel end-MU-field-lemmas)
(deftheory MU-field-lemmas
    (set-difference-theories
     (universal-theory 'end-MU-field-lemmas)
     (universal-theory 'begin-MU-field-lemmas)))
(deftheory MU-lch2-field-INST-at-lemmas
    '(MU-lch2-tag-=-inst-tag-2 MU-lch2-data-=-ML2-output-ML2-output-2))
(in-theory (disable MU-lch2-field-INST-at-lemmas))

(deftheory MU-field-INST-at-lemmas
    (union-theories (theory 'MU-RS-field-INST-at-lemmas)
    (union-theories (theory 'MU-lch1-field-INST-at-lemmas)
                    (theory 'MU-lch2-field-INST-at-lemmas))))

;;;;;;;;;;;;;;;;;;;;;; LSU field lemmas;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(deflabel begin-LSU-field-lemmas)

(deflabel begin-LSU-RS-field-lemmas)

(defthm LSU-RS0-valid-if-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(LSU RS0))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (LSU-RS-valid? (LSU-RS0 (MA-LSU MA))) 1))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-RS1-valid-if-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(LSU RS1))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (LSU-RS-valid? (LSU-RS1 (MA-LSU MA))) 1))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm uniq-inst-at-LSU-RS0-if-valid
    (implies (and (inv MT MA)
                  (b1p (LSU-RS-valid? (LSU-rs0 (MA-LSU MA))))
                  (MA-state-p MA) (MAETT-p MT))
             (uniq-INST-at-stg '(LSU RS0) MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-LSU-stg-conflict))))

(defthm uniq-inst-at-LSU-RS1-if-valid
    (implies (and (inv MT MA)
```

```
                (b1p (LSU-RS-valid? (LSU-rs1 (MA-LSU MA))))
                (MA-state-p MA) (MAETT-p MT))
           (uniq-INST-at-stg '(LSU RS1) MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-LSU-stg-conflict))))

(defthm no-inst-at-LSU-RS0
    (implies (and (inv MT MA)
                (not (b1p (LSU-RS-valid? (LSU-rs0 (MA-LSU MA)))))
                (MA-state-p MA) (MAETT-p MT))
           (no-INST-at-stg '(LSU RS0) MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-LSU-stg-conflict))))

(defthm no-inst-at-LSU-RS1
    (implies (and (inv MT MA)
                (not (b1p (LSU-RS-valid? (LSU-rs1 (MA-LSU MA)))))
                (MA-state-p MA) (MAETT-p MT))
           (no-INST-at-stg '(LSU RS1) MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-LSU-stg-conflict))))

(defthm not-INST-excpt-detected-p-if-LSU-RS0
    (implies (and (inv MT MA)
                (equal (INST-stg i) '(LSU RS0))
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (INST-in i MT)
                (MAETT-p MT) (MA-state-p MA))
           (not (INST-excpt-detected-p i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                :use (:instance INST-inv-if-INST-in))))

(defthm not-INST-excpt-detected-p-if-LSU-RS1
    (implies (and (inv MT MA)
                (equal (INST-stg i) '(LSU RS1))
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (INST-in i MT)
                (MAETT-p MT) (MA-state-p MA))
           (not (INST-excpt-detected-p i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                :use (:instance INST-inv-if-INST-in))))

(defthm not-LSU-RS0-op-if-not-LSU-RS-rdy1
    (implies (and (inv MT MA)
                (not (b1p (LSU-RS-rdy1? (LSU-RS0 (MA-LSU MA)))))
                (INST-in i MT)
                (equal (INST-stg i) '(LSU RS0))
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (MAETT-p MT) (MA-state-p MA))
           (equal (INST-LSU-op? i) 0))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                :use (:instance INST-inv-if-INST-in))))

(defthm not-LSU-RS1-op-if-not-LSU-RS-rdy1
    (implies (and (inv MT MA)
                (not (b1p (LSU-RS-rdy1? (LSU-RS1 (MA-LSU MA)))))
```

```
                (INST-in i MT)
                (equal (INST-stg i) '(LSU RS1))
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (MAETT-p MT) (MA-state-p MA))
           (equal (INST-LSU-op? i) 0))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                :use (:instance INST-inv-if-INST-in))))

(defthm LSU-RS0-op-=-inst-LSU-op
    (implies (and (inv MT MA)
                (INST-in i MT)
                (equal (INST-stg i) '(LSU RS0))
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (MAETT-p MT) (MA-state-p MA))
           (equal (LSU-RS-op (LSU-RS0 (MA-LSU MA)))
                  (INST-LSU-op? i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                :use (:instance INST-inv-if-INST-in))))

(defthm LSU-RS0-op-=-inst-LSU-op-2
    (implies (and (inv MT MA)
                (uniq-inst-at-stg '(LSU RS0) MT)
                (not (b1p (inst-specultv?
                           (inst-at-stg '(LSU RS0) MT))))
                (not (b1p (INST-modified?
                           (inst-at-stg '(LSU RS0) MT))))
                (MAETT-p MT) (MA-state-p MA))
           (equal (LSU-RS-op (LSU-RS0 (MA-LSU MA)))
                  (INST-LSU-op? (inst-at-stg '(LSU RS0) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                :use (:instance INST-inv-if-INST-in
                                (i (inst-at-stg '(LSU RS0) MT))))))

(defthm LSU-RS1-op-=-inst-LSU-op
    (implies (and (inv MT MA)
                (INST-in i MT)
                (equal (INST-stg i) '(LSU RS1))
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (MAETT-p MT) (MA-state-p MA))
           (equal (LSU-RS-op (LSU-RS1 (MA-LSU MA)))
                  (INST-LSU-op? i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                :use (:instance INST-inv-if-INST-in))))

(defthm LSU-RS1-op-=-inst-LSU-op-2
    (implies (and (inv MT MA)
                (uniq-inst-at-stg '(LSU RS1) MT)
                (not (b1p (inst-specultv?
                           (inst-at-stg '(LSU RS1) MT))))
                (not (b1p (INST-modified?
                           (inst-at-stg '(LSU RS1) MT))))
                (MAETT-p MT) (MA-state-p MA))
           (equal (LSU-RS-op (LSU-RS1 (MA-LSU MA)))
                  (INST-LSU-op? (inst-at-stg '(LSU RS1) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
```

```
                                (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (inst-at-stg '(LSU RS1) MT))))))

(defthm LSU-RS0-ld-st-=-inst-ld-st
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(LSU RS0))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-RS-ld-st? (LSU-RS0 (MA-LSU MA)))
                    (INST-ld-st? i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-RS0-ld-st-=-inst-ld-st-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(LSU RS0) MT)
                  (not (b1p (inst-specultv?
                             (inst-at-stg '(LSU RS0) MT))))
                  (not (b1p (INST-modified?
                             (inst-at-stg '(LSU RS0) MT))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-RS-ld-st? (LSU-RS0 (MA-LSU MA)))
                    (INST-ld-st? (inst-at-stg '(LSU RS0) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (inst-at-stg '(LSU RS0) MT))))))

(defthm LSU-RS1-ld-st-=-inst-ld-st
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(LSU RS1))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-RS-ld-st? (LSU-RS1 (MA-LSU MA)))
                    (INST-ld-st? i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-RS1-ld-st-=-inst-ld-st-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(LSU RS1) MT)
                  (not (b1p (inst-specultv?
                             (inst-at-stg '(LSU RS1) MT))))
                  (not (b1p (INST-modified?
                             (inst-at-stg '(LSU RS1) MT))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-RS-ld-st? (LSU-RS1 (MA-LSU MA)))
                    (INST-ld-st? (inst-at-stg '(LSU RS1) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (inst-at-stg '(LSU RS1) MT))))))

(defthm LSU-RS0-tag-=-inst-tag
    (implies (and (inv MT MA)
```

```
                (INST-in i MT)
                (equal (INST-stg i) '(LSU RS0))
                (MAETT-p MT) (MA-state-p MA))
           (equal (LSU-RS-tag (LSU-RS0 (MA-LSU MA)))
                  (INST-tag i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-RS0-tag-=-inst-tag-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(LSU RS0) MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-RS-tag (LSU-RS0 (MA-LSU MA)))
                    (INST-tag (INST-at-stg '(LSU RS0) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(LSU RS0) MT))))))

(defthm LSU-RS-tag-=-inst-tag
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(LSU RS1))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-RS-tag (LSU-RS1 (MA-LSU MA)))
                    (INST-tag i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-RS-tag-=-inst-tag-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(LSU RS1) MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-RS-tag (LSU-RS1 (MA-LSU MA)))
                    (INST-tag (INST-at-stg '(LSU RS1) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(LSU RS1) MT))))))

(defthm LSU-RS0-val1-=-INST-src-val1
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(LSU RS0))
                  (b1p (LSU-RS-rdy1? (LSU-RS0 (MA-LSU MA))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-RS-val1 (LSU-RS0 (MA-LSU MA))) (INST-src-val1 i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-RS0-val1-=-INST-src-val1-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(LSU RS0) MT)
                  (b1p (LSU-RS-rdy1? (LSU-RS0 (MA-LSU MA))))
                  (not (b1p (inst-specultv?
                             (INST-at-stg '(LSU RS0) MT))))
                  (not (b1p (INST-modified?
```

```
                          (INST-at-stg '(LSU RS0) MT))))
                (MAETT-p MT) (MA-state-p MA))
           (equal (LSU-RS-val1 (LSU-RS0 (MA-LSU MA)))
                  (INST-src-val1 (INST-at-stg '(LSU RS0) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(LSU RS0) MT))))))

(defthm LSU-RS1-val1-=-INST-src-val1
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(LSU RS1))
                  (b1p (LSU-RS-rdy1? (LSU-RS1 (MA-LSU MA))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-RS-val1 (LSU-RS1 (MA-LSU MA))) (INST-src-val1 i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-RS1-val1-=-INST-src-val1-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(LSU RS1) MT)
                  (b1p (LSU-RS-rdy1? (LSU-RS1 (MA-LSU MA))))
                  (not (b1p (inst-specultv?
                             (INST-at-stg '(LSU RS1) MT))))
                  (not (b1p (INST-modified?
                             (INST-at-stg '(LSU RS1) MT))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-RS-val1 (LSU-RS1 (MA-LSU MA)))
                    (INST-src-val1 (INST-at-stg '(LSU RS1) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(LSU RS1) MT))))))

(defthm LSU-RS0-val2-=-INST-src-val2
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(LSU RS0))
                  (b1p (LSU-RS-rdy2? (LSU-RS0 (MA-LSU MA))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-RS-val2 (LSU-RS0 (MA-LSU MA))) (INST-src-val2 i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-RS0-val2-=-INST-src-val2-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(LSU RS0) MT)
                  (b1p (LSU-RS-rdy2? (LSU-RS0 (MA-LSU MA))))
                  (not (b1p (inst-specultv?
                             (INST-at-stg '(LSU RS0) MT))))
                  (not (b1p (INST-modified?
                             (INST-at-stg '(LSU RS0) MT))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-RS-val2 (LSU-RS0 (MA-LSU MA)))
                    (INST-src-val2 (INST-at-stg '(LSU RS0) MT))))
```

```
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(LSU RS0) MT))))))

(defthm LSU-RS1-val2-=-INST-src-val2
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(LSU RS1))
                  (b1p (LSU-RS-rdy2? (LSU-RS1 (MA-LSU MA))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-RS-val2 (LSU-RS1 (MA-LSU MA))) (INST-src-val2 i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-RS1-val2-=-INST-src-val2-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(LSU RS1) MT)
                  (b1p (LSU-RS-rdy2? (LSU-RS1 (MA-LSU MA))))
                  (not (b1p (inst-specultv?
                             (INST-at-stg '(LSU RS1) MT))))
                  (not (b1p (INST-modified?
                             (INST-at-stg '(LSU RS1) MT))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-RS-val2 (LSU-RS1 (MA-LSU MA)))
                    (INST-src-val2 (INST-at-stg '(LSU RS1) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(LSU RS1) MT))))))

(defthm LSU-RS0-val3-=-INST-src-val3
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(LSU RS0))
                  (b1p (LSU-RS-rdy3? (LSU-RS0 (MA-LSU MA))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-RS-val3 (LSU-RS0 (MA-LSU MA))) (INST-src-val3 i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-RS0-val3-=-INST-src-val3-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(LSU RS0) MT)
                  (b1p (LSU-RS-rdy3? (LSU-RS0 (MA-LSU MA))))
                  (not (b1p (inst-specultv?
                             (INST-at-stg '(LSU RS0) MT))))
                  (not (b1p (INST-modified?
                             (INST-at-stg '(LSU RS0) MT))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-RS-val3 (LSU-RS0 (MA-LSU MA)))
                    (INST-src-val3 (INST-at-stg '(LSU RS0) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(LSU RS0) MT))))))
```

```
(defthm LSU-RS1-val3-=-INST-src-val3
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(LSU RS1))
                  (b1p (LSU-RS-rdy3? (LSU-RS1 (MA-LSU MA))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-RS-val3 (LSU-RS1 (MA-LSU MA))) (INST-src-val3 i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-RS1-val3-=-INST-src-val3-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(LSU RS1) MT)
                  (b1p (LSU-RS-rdy3? (LSU-RS1 (MA-LSU MA))))
                  (not (b1p (inst-specultv?
                             (INST-at-stg '(LSU RS1) MT))))
                  (not (b1p (INST-modified?
                             (INST-at-stg '(LSU RS1) MT))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-RS-val3 (LSU-RS1 (MA-LSU MA)))
                    (INST-src-val3 (INST-at-stg '(LSU RS1) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(LSU RS1) MT))))))

(deflabel end-LSU-RS-field-lemmas)
(deftheory LSU-RS-field-inst-at-lemmas
    '(LSU-RS0-op-=-inst-LSU-op-2 LSU-RS1-op-=-inst-LSU-op-2
      LSU-RS0-ld-st-=-inst-ld-st-2 LSU-RS1-ld-st-=-inst-ld-st-2
      LSU-RS0-tag-=-inst-tag-2 LSU-RS-tag-=-inst-tag-2
      LSU-RS0-val1-=-INST-src-val1-2 LSU-RS1-val1-=-INST-src-val1-2
      LSU-RS0-val2-=-INST-src-val2-2 LSU-RS1-val2-=-INST-src-val2-2
      LSU-RS0-val3-=-INST-src-val3-2 LSU-RS1-val3-=-INST-src-val3-2))
(in-theory (disable LSU-RS-field-inst-at-lemmas))

(deflabel begin-LSU-wbuf-field-lemmas)
(defthm LSU-wbuf0-valid-if-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (wbuf0-stg-p (INST-stg i))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))) 1))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter
                                                wbuf0-stg-p)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-wbuf1-valid-if-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (wbuf1-stg-p (INST-stg i))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))) 1))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter
                                                wbuf1-stg-p)
```

```
                                (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm uniq-inst-at-LSU-wbuf0-if-valid
    (implies (and (inv MT MA)
                  (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
                  (MA-state-p MA) (MAETT-p MT))
             (uniq-INST-at-stgs '((LSU wbuf0) (LSU wbuf0 lch)
                                  (complete wbuf0) (commit wbuf0))
                                MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-LSU-stg-conflict))))

(defthm uniq-inst-at-LSU-wbuf1-if-valid
    (implies (and (inv MT MA)
                  (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                  (MA-state-p MA) (MAETT-p MT))
             (uniq-INST-at-stgs '((LSU wbuf1) (LSU wbuf1 lch)
                                  (complete wbuf1) (commit wbuf1))
                                MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-LSU-stg-conflict))))

(defthm no-inst-at-LSU-wbuf0
    (implies (and (inv MT MA)
                  (not (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
                  (MA-state-p MA) (MAETT-p MT))
             (no-INST-at-stgs '((LSU wbuf0) (LSU wbuf0 lch)
                                (complete wbuf0) (commit wbuf0))
                              MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-LSU-stg-conflict))))

(defthm no-inst-at-LSU-wbuf1
    (implies (and (inv MT MA)
                  (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                  (MA-state-p MA) (MAETT-p MT))
             (no-INST-at-stgs '((LSU wbuf1) (LSU wbuf1 lch)
                                (complete wbuf1) (commit wbuf1))
                              MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-LSU-stg-conflict))))
(encapsulate nil
(local
(defthm uniq-wbuf0-inst-help-help
    (implies (and (member-equal i trace)
                  (wbuf0-stg-p (INST-stg i)))
             (not (no-inst-at-stgs-in-trace '((LSU wbuf0)
                                              (LSU wbuf0 lch)
                                              (complete wbuf0)
                                              (commit wbuf0)) trace)))
  :hints (("goal" :in-theory (enable wbuf0-stg-p)))))

(local
(defthm uniq-wbuf0-inst-help
    (implies (and (uniq-inst-at-stgs-in-trace '((LSU wbuf0)
                                                (LSU wbuf0 lch)
                                                (complete wbuf0)
                                                (commit wbuf0)) trace)
                  (member-equal i trace)
                  (member-equal j trace)
                  (wbuf0-stg-p (INST-stg i)) (wbuf0-stg-p (INST-stg j)))
```

```
            (equal i j))
  :hints (("Goal" :in-theory (enable wbuf0-stg-p)))
  :rule-classes nil))

(defthm uniq-wbuf0-inst
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (MAETT-p MT) (MA-state-p MA)
                  (wbuf0-stg-p (INST-stg i)) (wbuf0-stg-p (INST-stg j)))
             (equal i j))
  :hints (("goal" :use ((:instance uniq-wbuf0-inst-help
                                   (trace (MT-trace MT)))
                        (:instance UNIQ-INST-AT-LSU-WBUF0-IF-VALID))
                  :in-theory (enable uniq-inst-at-stgs INST-in)
                  :restrict ((LSU-WBUF0-VALID-IF-INST-IN ((i i))))))
  :rule-classes nil)
)

(encapsulate nil
(local
(defthm uniq-wbuf1-inst-help-help
    (implies (and (member-equal i trace)
                  (wbuf1-stg-p (INST-stg i)))
             (not (no-inst-at-stgs-in-trace '((LSU wbuf1)
                                              (LSU wbuf1 lch)
                                              (complete wbuf1)
                                              (commit wbuf1)) trace)))
  :hints (("goal" :in-theory (enable wbuf1-stg-p)))))

(local
(defthm uniq-wbuf1-inst-help
    (implies (and (uniq-inst-at-stgs-in-trace '((LSU wbuf1)
                                                (LSU wbuf1 lch)
                                                (complete wbuf1)
                                                (commit wbuf1)) trace)
                  (member-equal i trace)
                  (member-equal j trace)
                  (wbuf1-stg-p (INST-stg i)) (wbuf1-stg-p (INST-stg j)))
             (equal i j))
  :hints (("Goal" :in-theory (enable wbuf1-stg-p)))
  :rule-classes nil))

(defthm uniq-wbuf1-inst
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (wbuf1-stg-p (INST-stg i)) (wbuf1-stg-p (INST-stg j))
                  (MAETT-p MT) (MA-state-p MA))
             (equal i j))
  :hints (("goal" :use ((:instance uniq-wbuf1-inst-help
                                   (trace (MT-trace MT)))
                        (:instance UNIQ-INST-AT-LSU-WBUF1-IF-VALID))
                  :in-theory (enable uniq-inst-at-stgs INST-in)
                  :restrict ((LSU-WBUF1-VALID-IF-INST-IN ((i i))))))
  :rule-classes nil)
)

(defthm LSU-wbuf0-complete-if-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(LSU wbuf0))
```

```
                (MAETT-p MT) (MA-state-p MA)
                (INST-p i))
           (equal (wbuf-complete? (LSU-wbuf0 (MA-LSU MA))) 0))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-wbuf1-complete-if-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(LSU wbuf1))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (wbuf-complete? (LSU-wbuf1 (MA-LSU MA))) 0))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-wbuf0-commit-if-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (or (equal (INST-stg i) '(LSU wbuf0))
                      (equal (INST-stg i) '(LSU wbuf0 lch))
                      (equal (INST-stg i) '(complete wbuf0)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (wbuf-commit? (LSU-wbuf0 (MA-LSU MA))) 0))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-wbuf1-commit-if-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (or (equal (INST-stg i) '(LSU wbuf1))
                      (equal (INST-stg i) '(LSU wbuf1 lch))
                      (equal (INST-stg i) '(complete wbuf1)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (wbuf-commit? (LSU-wbuf1 (MA-LSU MA))) 0))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-wbuf1-complete-if-inst-at-LSU-wbuf0
    (implies (and (inv MT MA)
                  (uniq-INST-at-stg '(LSU wbuf0) MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (wbuf-commit? (LSU-wbuf0 (MA-LSU MA))) 0))
  :hints (("goal" :restrict
                  ((LSU-wbuf0-commit-if-INST-in
                    ((i (INST-at-stg '(LSU wbuf0) MT))))))))

(defthm LSU-wbuf1-complete-if-inst-at-LSU-wbuf0-lch
    (implies (and (inv MT MA)
                  (uniq-INST-at-stg '(LSU wbuf0 lch) MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (wbuf-commit? (LSU-wbuf0 (MA-LSU MA))) 0))
  :hints (("goal" :restrict
                  ((LSU-wbuf0-commit-if-INST-in
                    ((i (INST-at-stg '(LSU wbuf0 lch) MT))))))))
```

```
(defthm LSU-wbuf1-complete-if-inst-at-complete-wbuf0
    (implies (and (inv MT MA)
                  (uniq-INST-at-stg '(complete wbuf0) MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (wbuf-commit? (LSU-wbuf0 (MA-LSU MA))) 0))
  :hints (("goal" :restrict
                  ((LSU-wbuf0-commit-if-INST-in
                    ((i (INST-at-stg '(complete wbuf0) MT))))))))

(defthm LSU-wbuf1-complete-if-inst-at-LSU-wbuf1
    (implies (and (inv MT MA)
                  (uniq-INST-at-stg '(LSU wbuf1) MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (wbuf-commit? (LSU-wbuf1 (MA-LSU MA))) 0))
  :hints (("goal" :restrict
                  ((LSU-wbuf1-commit-if-INST-in
                    ((i (INST-at-stg '(LSU wbuf1) MT))))))))

(defthm LSU-wbuf1-complete-if-inst-at-LSU-wbuf1-lch
    (implies (and (inv MT MA)
                  (uniq-INST-at-stg '(LSU wbuf1 lch) MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (wbuf-commit? (LSU-wbuf1 (MA-LSU MA))) 0))
  :hints (("goal" :restrict
                  ((LSU-wbuf1-commit-if-INST-in
                    ((i (INST-at-stg '(LSU wbuf1 lch) MT))))))))

(defthm LSU-wbuf1-complete-if-inst-at-complete-wbuf1
    (implies (and (inv MT MA)
                  (uniq-INST-at-stg '(complete wbuf1) MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (wbuf-commit? (LSU-wbuf1 (MA-LSU MA))) 0))
  :hints (("goal" :restrict
                  ((LSU-wbuf1-commit-if-INST-in
                    ((i (INST-at-stg '(complete wbuf1) MT))))))))


(defthm LSU-wbuf0-=-inst-tag
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (or (equal (INST-stg i) '(LSU wbuf0))
                      (equal (INST-stg i) '(LSU wbuf0 lch))
                      (equal (INST-stg i) '(complete wbuf0)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (wbuf-tag (LSU-wbuf0 (MA-LSU MA)))
                    (INST-tag i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-wbuf1-=-inst-tag
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (or (equal (INST-stg i) '(LSU wbuf1))
                      (equal (INST-stg i) '(LSU wbuf1 lch))
                      (equal (INST-stg i) '(complete wbuf1)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (wbuf-tag (LSU-wbuf1 (MA-LSU MA)))
                    (INST-tag i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))
```

```
(defthm not-INST-excpt-detected-p-if-LSU-wbuf0
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(LSU wbuf0))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA))
             (not (INST-excpt-detected-p i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm not-INST-excpt-detected-p-if-LSU-wbuf1
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(LSU wbuf1))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA))
             (not (INST-excpt-detected-p i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-wbuf0-addr-=-INST-store-addr
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (wbuf0-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (wbuf-addr (LSU-wbuf0 (MA-LSU MA)))
                    (INST-store-addr i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter
                                                wbuf0-stg-p)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-wbuf1-addr-=-INST-store-addr
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (wbuf1-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA)))
                    (INST-store-addr i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter
                                                wbuf1-stg-p)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-wbuf0-addr-=-INST-store-addr-2
    (let ((i (INST-at-stgs '((LSU wbuf0) (LSU wbuf0 lch)
                             (complete wbuf0) (commit wbuf0))
                           MT)))
      (implies (and (inv MT MA)
                    (uniq-inst-at-stgs  '((LSU wbuf0)
                                          (LSU wbuf0 lch)
                                          (complete wbuf0)
                                          (commit wbuf0)) MT)
```

```
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (wbuf-addr (LSU-wbuf0 (MA-LSU MA)))
                    (INST-store-addr i))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use ((:instance INST-inv-if-INST-in
                                   (i (INST-at-stgs '((LSU wbuf0)
                                                      (LSU wbuf0 lch)
                                                      (complete wbuf0)
                                                      (commit wbuf0))
                                                    MT)))
                        (:instance INST-is-at-one-of-the-stages
                                   (i (INST-at-stgs '((LSU wbuf0)
                                                      (LSU wbuf0 lch)
                                                      (complete wbuf0)
                                                      (commit wbuf0))
                                                    MT)))))))

(defthm LSU-wbuf1-addr-=-INST-store-addr-2
    (let ((i (INST-at-stgs '((LSU wbuf1) (LSU wbuf1 lch)
                             (complete wbuf1) (commit wbuf1))
                           MT)))
      (implies (and (inv MT MA)
                    (uniq-inst-at-stgs  '((LSU wbuf1)
                                          (LSU wbuf1 lch)
                                          (complete wbuf1)
                                          (commit wbuf1)) MT)
                    (not (b1p (inst-specultv? i)))
                    (not (b1p (INST-modified? i)))
                    (MAETT-p MT) (MA-state-p MA))
               (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA)))
                      (INST-store-addr i))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use ((:instance INST-inv-if-INST-in
                                   (i (INST-at-stgs '((LSU wbuf1)
                                                      (LSU wbuf1 lch)
                                                      (complete wbuf1)
                                                      (commit wbuf1))
                                                    MT)))
                        (:instance INST-is-at-one-of-the-stages
                                   (i (INST-at-stgs '((LSU wbuf1)
                                                      (LSU wbuf1 lch)
                                                      (complete wbuf1)
                                                      (commit wbuf1))
                                                    MT)))))))

(defthm LSU-wbuf0-val-=-INST-src-val3
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (wbuf0-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (wbuf-val (LSU-wbuf0 (MA-LSU MA)))
                    (INST-src-val3 i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter
                                                wbuf0-stg-p)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))
```

```
(defthm LSU-wbuf1-val-=-INST-src-val3
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (wbuf1-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (wbuf-val (LSU-wbuf1 (MA-LSU MA)))
                    (INST-src-val3 i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter
                                                wbuf1-stg-p)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-wbuf0-val-=-INST-src-val3-2
    (let ((i (INST-at-stgs '((LSU wbuf0) (LSU wbuf0 lch)
                             (complete wbuf0) (commit wbuf0))
                           MT)))
      (implies (and (inv MT MA)
                    (uniq-inst-at-stgs  '((LSU wbuf0)
                                          (LSU wbuf0 lch)
                                          (complete wbuf0)
                                          (commit wbuf0)) MT)
                    (not (b1p (inst-specultv? i)))
                    (not (b1p (INST-modified? i)))
                    (MAETT-p MT) (MA-state-p MA))
               (equal (wbuf-val (LSU-wbuf0 (MA-LSU MA)))
                      (INST-src-val3 i))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use ((:instance INST-inv-if-INST-in
                                   (i (INST-at-stgs '((LSU wbuf0)
                                                      (LSU wbuf0 lch)
                                                      (complete wbuf0)
                                                      (commit wbuf0))
                                                    MT)))
                        (:instance INST-is-at-one-of-the-stages
                                   (i (INST-at-stgs '((LSU wbuf0)
                                                      (LSU wbuf0 lch)
                                                      (complete wbuf0)
                                                      (commit wbuf0))
                                                    MT)))))))

(defthm LSU-wbuf1-val-=-INST-src-val3-2
    (let ((i (INST-at-stgs '((LSU wbuf1) (LSU wbuf1 lch)
                             (complete wbuf1) (commit wbuf1))
                           MT)))
      (implies (and (inv MT MA)
                    (uniq-inst-at-stgs  '((LSU wbuf1)
                                          (LSU wbuf1 lch)
                                          (complete wbuf1)
                                          (commit wbuf1)) MT)
                    (not (b1p (inst-specultv? i)))
                    (not (b1p (INST-modified? i)))
                    (MAETT-p MT) (MA-state-p MA))
               (equal (wbuf-val (LSU-wbuf1 (MA-LSU MA)))
                      (INST-src-val3 i))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use ((:instance INST-inv-if-INST-in
                                   (i (INST-at-stgs '((LSU wbuf1)
```

```
                                                  (LSU wbuf1 lch)
                                                  (complete wbuf1)
                                                  (commit wbuf1))
                                                MT)))
                          (:instance INST-is-at-one-of-the-stages
                                     (i (INST-at-stgs '((LSU wbuf1)
                                                  (LSU wbuf1 lch)
                                                  (complete wbuf1)
                                                  (commit wbuf1))
                                                MT)))))))

(deflabel end-LSU-wbuf-field-lemmas)
(deftheory LSU-wbuf-field-inst-at-lemmas
    '(LSU-wbuf0-addr-=-INST-store-addr-2 LSU-wbuf1-addr-=-INST-store-addr-2
      LSU-wbuf0-val-=-INST-src-val3-2 LSU-wbuf1-val-=-INST-src-val3-2))

(deflabel begin-LSU-rbuf-field-lemmas)
(defthm LSU-rbuf-valid-if-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(LSU rbuf))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (rbuf-valid? (LSU-rbuf (MA-LSU MA))) 1))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm uniq-inst-at-LSU-rbuf-if-valid
    (implies (and (inv MT MA)
                  (b1p (rbuf-valid? (LSU-rbuf (MA-LSU MA))))
                  (MA-state-p MA) (MAETT-p MT))
             (uniq-INST-at-stg '(LSU rbuf) MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-LSU-stg-conflict))))
(defthm no-inst-at-LSU-rbuf
    (implies (and (inv MT MA)
                  (not (b1p (rbuf-valid? (LSU-rbuf (MA-LSU MA)))))
                  (MA-state-p MA) (MAETT-p MT))
             (no-INST-at-stg '(LSU rbuf) MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-LSU-stg-conflict))))

(defthm LSU-rbuf-tag-=-inst-tag
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(LSU rbuf))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (rbuf-tag (LSU-rbuf (MA-LSU MA)))
                    (INST-tag i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm not-INST-excpt-detected-p-if-LSU-rbuf
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA))
             (not (INST-excpt-detected-p i)))
```

```
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-rbuf-addr-=-INST-load-addr
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (rbuf-addr (LSU-rbuf (MA-LSU MA)))
                    (INST-load-addr i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(deflabel end-LSU-rbuf-field-lemmas)

(defthm LSU-lch-valid-if-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (or (equal (INST-stg i) '(LSU lch))
                      (equal (INST-stg i) '(LSU wbuf0 lch))
                      (equal (INST-stg i) '(LSU wbuf1 lch)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (LSU-latch-valid? (LSU-lch (MA-LSU MA))) 1))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm uniq-inst-at-LSU-lch-if-valid
    (implies (and (inv MT MA)
                  (b1p (LSU-latch-valid? (LSU-lch (MA-LSU MA))))
                  (MA-state-p MA) (MAETT-p MT))
             (uniq-INST-at-stgs '((LSU lch) (LSU wbuf0 lch)
                                  (LSU wbuf1 lch))
                                MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-LSU-stg-conflict))))

(defthm no-inst-at-LSU-lch
    (implies (and (inv MT MA)
                  (not (b1p (LSU-latch-valid? (LSU-lch (MA-LSU MA)))))
                  (MA-state-p MA) (MAETT-p MT))
             (no-INST-at-stgs '((LSU lch) (LSU wbuf0 lch)
                                (LSU wbuf1 lch))
                              MT))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     no-LSU-stg-conflict))))

(defthm LSU-lch-=-inst-tag
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (or (equal (INST-stg i) '(LSU lch))
                      (equal (INST-stg i) '(LSU wbuf0 lch))
                      (equal (INST-stg i) '(LSU wbuf1 lch)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-latch-tag (LSU-lch (MA-LSU MA)))
                    (INST-tag i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
```

```
                                (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-lch-=-inst-tag-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stgs '((LSU lch) (LSU wbuf0 lch)
                                       (LSU wbuf1 lch))
                                     MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-latch-tag (LSU-lch (MA-LSU MA)))
                    (INST-tag (INST-at-stgs '((LSU lch)
                                              (LSU wbuf0 lch)
                                              (LSU wbuf1 lch)) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stgs '((LSU lch)
                                                     (LSU wbuf0 lch)
                                                     (LSU wbuf1 lch))
                                                   MT))))))

(defthm LSU-load-if-at-LSU-rbuf-lch
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (or (equal (INST-stg i) '(LSU rbuf))
                      (equal (INST-stg i) '(LSU lch)))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-load? i) 1))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))
(in-theory (disable LSU-load-if-at-LSU-rbuf-lch))

(defthm LSU-load-INST-at-LSU-lch-2
    (implies (and (inv MT MA)
                  (uniq-INST-at-stg '(LSU lch) MT)
                  (not (b1p (inst-specultv?
                             (INST-at-stg '(LSU lch) MT))))
                  (not (b1p (INST-modified?
                             (INST-at-stg '(LSU lch) MT))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-load? (INST-at-stg '(LSU lch) MT)) 1))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(LSU lch) MT))))))

(defthm LSU-store-if-at-LSU-wbuf
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (or (equal (INST-stg i) '(LSU wbuf0))
                      (equal (INST-stg i) '(LSU wbuf1))
                      (equal (INST-stg i) '(LSU wbuf0 lch))
                      (equal (INST-stg i) '(LSU wbuf1 lch))
                      (equal (INST-stg i) '(complete wbuf0))
                      (equal (INST-stg i) '(complete wbuf1))
                      (equal (INST-stg i) '(commit wbuf0))
                      (equal (INST-stg i) '(commit wbuf1)))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
```

```
                (MAETT-p MT) (MA-state-p MA))
           (equal (INST-store? i) 1))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))
(in-theory (disable LSU-store-if-at-LSU-wbuf))

(defthm LSU-lch-excpt-=-INST-excpt-flags
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (or (equal (INST-stg i) '(LSU lch))
                      (equal (INST-stg i) '(LSU wbuf0 lch))
                      (equal (INST-stg i) '(LSU wbuf1 lch)))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-latch-excpt (LSU-lch (MA-LSU MA)))
                    (INST-excpt-flags i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-lch-excpt-=-INST-excpt-flags-2
    (let ((i (INST-at-stgs '((LSU lch)
                             (LSU wbuf0 lch)
                             (LSU wbuf1 lch)) MT)))
      (implies (and (inv MT MA)
                    (uniq-inst-at-stgs '((LSU lch)
                                         (LSU wbuf0 lch)
                                         (LSU wbuf1 lch))
                                       MT)
                    (not (b1p (inst-specultv? i)))
                    (not (b1p (INST-modified? i)))
                    (MAETT-p MT) (MA-state-p MA))
               (equal (LSU-latch-tag (LSU-lch (MA-LSU MA)))
                      (INST-tag i))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stgs '((LSU lch)
                                                     (LSU wbuf0 lch)
                                                     (LSU wbuf1 lch))
                                                   MT))))))

(defthm LSU-latch-val-=-INST-dest-val
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(LSU lch))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-load-accs-error-detected-p i))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-latch-val (LSU-lch (MA-LSU MA)))
                    (INST-dest-val i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-latch-val-=-INST-dest-val-2
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(LSU lch) MT)
                  (not (b1p (inst-specultv?
```

```
                          (INST-at-stg '(LSU lch) MT))))
                (not (b1p (INST-modified?
                          (INST-at-stg '(LSU lch) MT))))
                (not (INST-load-accs-error-detected-p
                       (INST-at-stg '(LSU lch) MT)))
                (MAETT-p MT) (MA-state-p MA))
           (equal (LSU-latch-val (LSU-lch (MA-LSU MA)))
                  (INST-dest-val (INST-at-stg '(LSU lch) MT))))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in
                                  (i (INST-at-stg '(LSU lch) MT))))))

(deflabel end-LSU-field-lemmas)
(deftheory LSU-field-lemmas
    (set-difference-theories
     (universal-theory 'end-LSU-field-lemmas)
     (universal-theory 'begin-LSU-field-lemmas)))

(deftheory LSU-field-INST-at-lemmas
    '(LSU-lch-=-inst-tag-2 LSU-load-INST-at-LSU-lch-2
      LSU-lch-excpt-=-INST-excpt-flags-2
      LSU-latch-val-=-INST-dest-val-2))
(in-theory (disable LSU-field-INST-at-lemmas))

;;;;;;;;;;;;;;;;Order of LSU instructions ;;;;;;;;;;;;;;;;
(defthm not-LSU-rbuf-wbuf0-if-not-wbuf-valid
    (implies (and (inv MT MA)
                  (b1p (rbuf-valid? (LSU-rbuf (MA-LSU MA))))
                  (b1p (rbuf-wbuf0? (LSU-rbuf (MA-LSU MA))))
                  (MAETT-p MT) (MA-state-p MA))
             (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (b1p (rbuf-valid? (LSU-rbuf (MA-LSU MA))))
                           (not (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
                           (MAETT-p MT) (MA-state-p MA))
                      (not (b1p (rbuf-wbuf0? (LSU-rbuf (MA-LSU MA))))))))
  :hints (("goal" :in-theory (enable inv consistent-MA-p
                                     consistent-LSU-p))))

(defthm not-LSU-rbuf-wbuf1-if-not-wbuf-valid
    (implies (and (inv MT MA)
                  (b1p (rbuf-valid? (LSU-rbuf (MA-LSU MA))))
                  (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA))))
                  (MAETT-p MT) (MA-state-p MA))
             (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (b1p (rbuf-valid? (LSU-rbuf (MA-LSU MA))))
                           (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                           (MAETT-p MT) (MA-state-p MA))
                      (not (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA))))))))
  :hints (("goal" :in-theory (enable inv consistent-MA-p
                                     consistent-LSU-p))))

(defthm in-order-wbuf0-rbuf-p-MT-trace
    (implies (and (inv MT MA)
```

```
                (MAETT-p MT) (MA-state-p MA)
                (b1p (rbuf-valid? (LSU-rbuf (MA-LSU MA))))
                (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
                (b1p (rbuf-wbuf0? (LSU-rbuf (MA-LSU MA)))))
           (in-order-wbuf0-rbuf-p (MT-trace MT)))
  :hints (("goal" :in-theory (enable inv in-order-LSU-inst-p
                                     in-order-load-store-p))))

(defthm in-order-rbuf-wbuf0-p-MT-trace
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (rbuf-valid? (LSU-rbuf (MA-LSU MA))))
                (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
                (not (b1p (rbuf-wbuf0? (LSU-rbuf (MA-LSU MA))))))
           (in-order-rbuf-wbuf0-p (MT-trace MT)))
  :hints (("goal" :in-theory (enable inv in-order-LSU-inst-p
                                     in-order-load-store-p))))

(defthm in-order-wbuf1-rbuf-p-MT-trace
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (rbuf-valid? (LSU-rbuf (MA-LSU MA))))
                (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA)))))
           (in-order-wbuf1-rbuf-p (MT-trace MT)))
  :hints (("goal" :in-theory (enable inv in-order-LSU-inst-p
                                     in-order-load-store-p))))

(defthm in-order-rbuf-wbuf1-p-MT-trace
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (rbuf-valid? (LSU-rbuf (MA-LSU MA))))
                (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                (not (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA))))))
           (in-order-rbuf-wbuf1-p (MT-trace MT)))
  :hints (("goal" :in-theory (enable inv in-order-LSU-inst-p
                                     in-order-load-store-p))))

(encapsulate nil
(local
 (defthm INST-in-order-p-LSU-RS1-RS0-help-help
     (implies (and (member-equal i trace)
                   (equal (INST-stg i) '(LSU RS1)))
              (not (no-inst-at-stg-in-trace '(LSU RS1) trace)))))

(local
(defthm INST-in-order-p-LSU-RS1-RS0-help
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (subtrace-p trace MT) (INST-listp trace)
                (IN-order-LSU-RS-p trace MA)
                (member-equal i trace)
                (equal (INST-stg j) '(LSU RS0))
                (member-equal j trace)
                (equal (INST-stg i) '(LSU RS1))
                (b1p (LSU-RS1-head? (MA-LSU MA))))
           (member-in-order i j trace))
  :hints (("Goal" :in-theory (enable member-in-order*)))))

(defthm INST-in-order-p-LSU-RS1-RS0
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
```

```
                (INST-in i MT) (INST-p i)
                (equal (INST-stg j) '(LSU RS0))
                (INST-in j MT) (INST-p j)
                (equal (INST-stg i) '(LSU RS1))
                (b1p (LSU-RS1-head? (MA-LSU MA))))
           (INST-in-order-p i j MT))
  :hints (("Goal" :in-theory (enable inv IN-ORDER-LSU-INST-P
                                     INST-in
                                     INST-in-order-p
                                     IN-ORDER-LSU-RS-P)
                :restrict ((INST-in-order-p-LSU-RS1-RS0-help
                            ((MT MT) (MA MA)))))))
)

(encapsulate nil
(local
 (defthm INST-in-order-p-LSU-RS0-RS1-help-help
     (implies (and (member-equal i trace)
                   (equal (INST-stg i) '(LSU RS0)))
              (not (no-inst-at-stg-in-trace '(LSU RS0) trace)))))

(local
(defthm INST-in-order-p-LSU-RS0-RS1-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (IN-order-LSU-RS-p trace MA)
                  (member-equal i trace)
                  (equal (INST-stg i) '(LSU RS0))
                  (member-equal j trace)
                  (equal (INST-stg j) '(LSU RS1))
                  (not (b1p (LSU-RS1-head? (MA-LSU MA)))))
             (member-in-order i j trace))
  :hints (("Goal" :in-theory (enable member-in-order*)))))

(defthm INST-in-order-p-LSU-RS0-RS1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU RS0))
                  (INST-in j MT) (INST-p j)
                  (equal (INST-stg j) '(LSU RS1))
                  (not (b1p (LSU-RS1-head? (MA-LSU MA)))))
             (INST-in-order-p i j MT))
  :hints (("Goal" :in-theory (enable inv IN-ORDER-LSU-INST-P
                                     IN-ORDER-LSU-RS-P
                                     INST-in-order-p INST-in )
                :restrict ((INST-in-order-p-LSU-RS0-RS1-help
                            ((MA MA) (MT MT)))))))
)

(in-theory (disable INST-in-order-p-LSU-RS1-RS0
                    INST-in-order-p-LSU-RS0-RS1))

(encapsulate nil
(local
 (defthm not-LSU-issued-stg-p-if-member-equal
     (implies (and (LSU-issued-stg-p (INST-stg i))
                   (no-issued-LSU-inst-p trace))
              (not (member-equal i trace)))))

(local
```

```
(defthm INST-in-order-p-LSU-issued-RS-help
    (implies (and (in-order-LSU-issue-p trace)
                  (member-equal i trace)
                  (LSU-issued-stg-p (INST-stg i))
                  (member-equal j trace)
                  (or (equal (INST-stg j) '(LSU RS0))
                      (equal (INST-stg j) '(LSU RS1))))
             (member-in-order i j trace))
  :hints (("goal" :in-theory (enable member-in-order*)))))

(defthm INST-in-order-p-LSU-issued-RS
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (LSU-issued-stg-p (INST-stg i))
                  (INST-in j MT) (INST-p j)
                  (or (equal (INST-stg j) '(LSU RS0))
                      (equal (INST-stg j) '(LSU RS1))))
             (inst-in-order-p i j MT))
  :hints (("goal" :in-theory (enable inst-in-order-p INST-in
                                     inv in-order-LSU-inst-p))))
)
(in-theory (disable INST-in-order-p-LSU-issued-RS))

(encapsulate nil
(local
(defthm not-no-inst-at-stg-in-trace-if-member-equal
    (implies (and (member-equal i trace)
                  (equal (INST-stg i) stg))
             (not (no-inst-at-stg-in-trace stg trace)))))

(local
(defthm not-no-inst-at-wbuf0-p-if-member-equal
    (implies (and (member-equal i trace)
                  (wbuf0-stg-p (INST-stg i)))
             (not (no-inst-at-wbuf0-p trace)))))

(local
(defthm not-no-inst-at-wbuf1-p-if-member-equal
    (implies (and (member-equal i trace)
                  (wbuf1-stg-p (INST-stg i)))
             (not (no-inst-at-wbuf1-p trace)))))

(local
(defthm INST-in-order-p-rbuf-wbuf0-help-help
    (implies (and (in-order-rbuf-wbuf0-p trace)
                  (member-equal i trace)
                  (equal (INST-stg i) '(LSU rbuf))
                  (member-equal j trace)
                  (wbuf0-stg-p (INST-stg j)))
             (member-in-order i j trace))
  :hints (("goal" :in-theory (enable member-in-order*)))))

(local
(defthm INST-in-order-p-rbuf-wbuf0-help
    (implies (and (inv MT MA)
                  (in-order-rbuf-wbuf0-p (MT-trace MT))
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU rbuf))
                  (INST-in j MT) (INST-p j)
                  (wbuf0-stg-p (INST-stg j))
                  (not (b1p (rbuf-wbuf0? (LSU-rbuf (MA-LSU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
```

```
            (INST-in-order-p i j MT))
  :hints (("goal" :in-theory (enable INST-in-order-p INST-in)))
  :rule-classes nil))

(defthm INST-in-order-p-rbuf-wbuf0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU rbuf))
                  (INST-in j MT) (INST-p j)
                  (wbuf0-stg-p (INST-stg j))
                  (not (b1p (rbuf-wbuf0? (LSU-rbuf (MA-LSU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (INST-in-order-p i j MT))
  :hints (("goal" :use (:instance INST-in-order-p-rbuf-wbuf0-help)
                  :restrict ((LSU-RBUF-VALID-IF-INST-IN
                              ((i i)))))))

(local
(defthm INST-in-order-p-wbuf0-rbuf-help-help
    (implies (and (in-order-wbuf0-rbuf-p trace)
                  (member-equal i trace)
                  (equal (INST-stg i) '(LSU rbuf))
                  (member-equal j trace)
                  (wbuf0-stg-p (INST-stg j)))
             (member-in-order j i trace))
  :hints (("goal" :in-theory (enable member-in-order*)))))

(local
(defthm INST-in-order-p-wbuf0-rbuf-help
    (implies (and (inv MT MA)
                  (in-order-wbuf0-rbuf-p (MT-trace MT))
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU rbuf))
                  (INST-in j MT) (INST-p j)
                  (wbuf0-stg-p (INST-stg j))
                  (b1p (rbuf-wbuf0? (LSU-rbuf (MA-LSU MA))))
                  (MAETT-p MT) (MA-state-p MA))
             (INST-in-order-p j i MT))
  :hints (("goal" :in-theory (enable INST-in-order-p INST-in)))
  :rule-classes nil))

(defthm INST-in-order-p-wbuf0-rbuf
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (wbuf0-stg-p (INST-stg i))
                  (INST-in j MT) (INST-p j)
                  (equal (INST-stg j) '(LSU rbuf))
                  (b1p (rbuf-wbuf0? (LSU-rbuf (MA-LSU MA))))
                  (MAETT-p MT) (MA-state-p MA))
             (INST-in-order-p i j MT))
  :hints (("goal" :use (:instance INST-in-order-p-wbuf0-rbuf-help
                                  (i j) (j i))
                  :restrict ((LSU-RBUF-VALID-IF-INST-IN
                              ((i j)))))))
(local
(defthm INST-in-order-p-rbuf-wbuf1-help-help
    (implies (and (in-order-rbuf-wbuf1-p trace)
                  (member-equal i trace)
                  (equal (INST-stg i) '(LSU rbuf))
                  (member-equal j trace)
                  (wbuf1-stg-p (INST-stg j)))
             (member-in-order i j trace))
```

```
  :hints (("goal" :in-theory (enable member-in-order*)))))

(local
(defthm INST-in-order-p-rbuf-wbuf1-help
    (implies (and (inv MT MA)
                  (in-order-rbuf-wbuf1-p (MT-trace MT))
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU rbuf))
                  (INST-in j MT) (INST-p j)
                  (wbuf1-stg-p (INST-stg j))
                  (not (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (INST-in-order-p i j MT))
  :hints (("goal" :in-theory (enable INST-in-order-p INST-in)))
  :rule-classes nil))

(defthm INST-in-order-p-rbuf-wbuf1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU rbuf))
                  (INST-in j MT) (INST-p j)
                  (wbuf1-stg-p (INST-stg j))
                  (not (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (INST-in-order-p i j MT))
  :hints (("goal" :use (:instance INST-in-order-p-rbuf-wbuf1-help)
                  :restrict ((LSU-RBUF-VALID-IF-INST-IN
                              ((i i)))))))

(local
(defthm INST-in-order-p-wbuf1-rbuf-help-help
    (implies (and (in-order-wbuf1-rbuf-p trace)
                  (member-equal i trace)
                  (equal (INST-stg i) '(LSU rbuf))
                  (member-equal j trace)
                  (wbuf1-stg-p (INST-stg j)))
             (member-in-order j i trace))
  :hints (("goal" :in-theory (enable member-in-order*)))))

(local
(defthm INST-in-order-p-wbuf1-rbuf-help
    (implies (and (inv MT MA)
                  (in-order-wbuf1-rbuf-p (MT-trace MT))
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU rbuf))
                  (INST-in j MT) (INST-p j)
                  (wbuf1-stg-p (INST-stg j))
                  (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA))))
                  (MAETT-p MT) (MA-state-p MA))
             (INST-in-order-p j i MT))
  :hints (("goal" :in-theory (enable INST-in-order-p INST-in)))
  :rule-classes nil))

(defthm INST-in-order-p-wbuf1-rbuf
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (wbuf1-stg-p (INST-stg i))
                  (INST-in j MT) (INST-p j)
                  (equal (INST-stg j) '(LSU rbuf))
                  (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA))))
                  (MAETT-p MT) (MA-state-p MA))
             (INST-in-order-p i j MT))
```

```
  :hints (("goal" :use (:instance INST-in-order-p-wbuf1-rbuf-help
                                  (i j) (j i))
                  :restrict ((LSU-RBUF-VALID-IF-INST-IN
                              ((i j)))))))
)

(in-theory (disable INST-in-order-p-wbuf0-rbuf
                    INST-in-order-p-rbuf-wbuf0
                    INST-in-order-p-wbuf1-rbuf
                    INST-in-order-p-rbuf-wbuf1))

(encapsulate nil
(local
 (defthm not-no-inst-at-wbuf0-if-member-equal
     (implies (and (member-equal i trace)
                   (wbuf0-stg-p (INST-stg i)))
              (not (no-inst-at-wbuf0-p trace)))))

(local
(defthm INST-in-order-p-wbuf0-wbuf1-help
    (implies (and (distinct-member-p trace)
                  (in-order-wb-trace-p trace)
                  (member-equal i trace)
                  (wbuf0-stg-p (INST-stg i))
                  (member-equal j trace)
                  (wbuf1-stg-p (INST-stg j)))
             (member-in-order i j trace))
  :hints (("goal" :in-theory (enable member-in-order*)))))

(defthm INST-in-order-p-wbuf0-wbuf1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (wbuf0-stg-p (INST-stg i))
                  (INST-in j MT) (INST-p j)
                  (wbuf1-stg-p (INST-stg j))
                  (MAETT-p MT) (MA-state-p MA))
             (INST-in-order-p i j MT))
  :hints (("goal" :in-theory (enable INST-in-order-p INST-in inv
                                     weak-inv MT-distinct-INST-p
                                     in-order-LSU-inst-p))))
)
(in-theory (disable INST-in-order-p-wbuf0-wbuf1))

(encapsulate nil
(local
 (defthm not-no-retired-stored-p-if-member-equal
     (implies (and (member-equal i trace)
                   (retire-stg-p (INST-stg i))
                   (b1p (INST-store? i)))
              (not (no-retired-store-p trace)))))

(local
(defthm INST-in-order-p-retire-wbuf0-help
    (implies (and (in-order-WB-retire-p trace)
                  (member-equal i trace)
                  (retire-stg-p (INST-stg i))
                  (b1p (INST-store? i))
                  (member-equal j trace)
                  (wbuf0-stg-p (INST-stg j)))
             (member-in-order i j trace))
  :hints (("goal" :in-theory (enable member-in-order*)))))
```

```
(defthm INST-in-order-p-retire-wbuf0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (retire-stg-p (INST-stg i))
                  (b1p (INST-store? i))
                  (INST-in j MT) (INST-p j)
                  (wbuf0-stg-p (INST-stg j)))
             (INST-in-order-p i j MT))
  :hints (("goal" :in-theory (enable inv in-order-LSU-inst-p
                                     INST-in INST-in-order-p))))
)
(in-theory (disable INST-in-order-p-retire-wbuf0))

(defthm INST-in-order-p-retire-wbuf1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (retire-stg-p (INST-stg i))
                  (b1p (INST-store? i))
                  (INST-in j MT) (INST-p j)
                  (wbuf1-stg-p (INST-stg j))
                  (MAETT-p MT) (MA-state-p MA))
             (INST-in-order-p i j MT))
  :hints (("goal" :restrict (((:rewrite INST-IN-ORDER-TRANSITIVITY . 1)
                              ((j (inst-at-stgs '((LSU WBUF0)
                                                  (LSU WBUF0 LCH)
                                                  (COMPLETE WBUF0)
                                                  (COMMIT WBUF0)) MT)))))
                  :in-theory (enable inst-in-order-p-retire-wbuf0
                                     inst-in-order-p-wbuf0-wbuf1))))
(in-theory (disable INST-in-order-p-retire-wbuf1))

(defthm INST-in-order-p-retired-store-non-retired
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (retire-stg-p (INST-stg i))
                  (b1p (INST-store? i))
                  (INST-in j MT) (INST-p j)
                  (not (retire-stg-p (INST-stg j)))
                  (MAETT-p MT) (MA-state-p MA))
             (INST-in-order-p i j MT))
  :hints (("goal" :use ((:instance INST-is-at-one-of-the-stages (i j)))
                  :in-theory (e/d (commit-stg-p
                                   INST-in-order-p-retire-wbuf0
                                   INST-in-order-p-retire-wbuf1)
                                  (INST-is-at-one-of-the-stages)))))

(in-theory (disable INST-in-order-p-retired-store-non-retired))

(encapsulate nil
(local
 (defthm no-retired-store-p-cdr-help
     (implies (and (in-order-wb-retire-p trace)
                   (tail-p sub trace)
                   (consp sub)
                   (equal (INST-stg (car sub)) '(commit wbuf0)))
              (no-retired-store-p (cdr sub)))))

(defthm no-retired-store-p-cdr
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (consp trace)
```

```
                (equal (INST-stg (car trace)) '(commit wbuf0)))
            (no-retired-store-p (cdr trace)))
  :hints (("goal" :in-theory (enable inv IN-ORDER-LSU-INST-P
                                     subtrace-p))))
)

(defthm retire-stg-p-car-if-member-inst-at-commit-wbuf0
    (implies (and (inv MT MA)
                  (member-equal i (cdr trace))
                  (INST-in i MT) (INST-p i)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (equal (INST-stg i) '(commit wbuf0))
                  (MAETT-p MT) (MA-state-p MA))
             (retire-stg-p (INST-stg (car trace))))
  :hints (("goal" :use ((:instance inst-is-at-one-of-the-stages
                                   (i (car trace)))
                        (:instance INST-IN-ORDER-P-WBUF0-WBUF1
                                   (i i) (j (car trace)))
                        (:instance INST-at-stg-inst-stg-commit (i i))
                        (:instance INST-at-stg-inst-stg-commit
                                   (i (car trace))))
                  :in-theory (enable commit-stg-p)
                  :do-not-induct t)))
;;; End of the theory of LSU instructions

;;;;;;;;;;;;;;;;;;complete field lemmas;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defthm LSU-wbuf0-complete-if-complete-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (or (equal (INST-stg i) '(LSU wbuf0 lch))
                      (equal (INST-stg i) '(complete wbuf0))
                      (equal (INST-stg i) '(commit wbuf0)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (wbuf-complete? (LSU-wbuf0 (MA-LSU MA))) 1))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-wbuf1-complete-if-complete-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (or (equal (INST-stg i) '(LSU wbuf1 lch))
                      (equal (INST-stg i) '(complete wbuf1))
                      (equal (INST-stg i) '(commit wbuf1)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (wbuf-complete? (LSU-wbuf1 (MA-LSU MA))) 1))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm not-INST-store-if-complete
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(complete))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i)))
```

```
            (equal (INST-store? i) 0))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))
(in-theory (disable not-INST-store-if-complete))
;;;;;;;;;;;;;;;;Commit field lemmas;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defthm LSU-wbuf0-commit-if-commit-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(commit wbuf0))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (wbuf-commit? (LSU-wbuf0 (MA-LSU MA))) 1))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-wbuf1-commit-if-commit-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(commit wbuf1))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (wbuf-commit? (LSU-wbuf1 (MA-LSU MA))) 1))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(defthm LSU-wbuf0-commit-if-commit-INST-in-2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg '(commit wbuf0) MT))
             (equal (wbuf-commit? (LSU-wbuf0 (MA-LSU MA))) 1))
  :hints (("Goal" :use (:instance LSU-WBUF0-COMMIT-IF-COMMIT-INST-IN
                                  (i (inst-at-stg '(commit wbuf0) MT)))
                  :in-theory (disable LSU-WBUF0-COMMIT-IF-COMMIT-INST-IN))))

(defthm LSU-wbuf1-commit-if-commit-INST-in-2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg '(commit wbuf1) MT))
             (equal (wbuf-commit? (LSU-wbuf1 (MA-LSU MA))) 1))
  :hints (("Goal" :use (:instance LSU-WBUF1-COMMIT-IF-COMMIT-INST-IN
                                  (i (inst-at-stg '(commit wbuf1) MT)))
                  :in-theory (disable LSU-WBUF1-COMMIT-IF-COMMIT-INST-IN))))

(defthm uniq-inst-at-stg-LSU-wbuf0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (wbuf-complete? (LSU-wbuf0 (MA-LSU MA)))))
                  (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
             (uniq-inst-at-stg '(LSU wbuf0) MT))
  :hints (("goal" :use
                  ((:instance UNIQ-INST-AT-LSU-WBUF0-IF-VALID)
                   (:instance uniq-inst-at-stgs*
                              (stgs '((LSU WBUF0)
                                      (LSU WBUF0 LCH)
                                      (COMPLETE WBUF0)
                                      (COMMIT WBUF0))))
                   (:instance uniq-inst-at-stgs*
                              (stgs '((LSU WBUF0 LCH)
                                      (COMPLETE WBUF0)
```

```
                                (COMMIT WBUF0))))
                 (:instance uniq-inst-at-stgs*
                            (stgs '((COMPLETE WBUF0)
                                    (COMMIT WBUF0))))
                 (:instance LSU-WBUF0-COMPLETE-IF-COMPLETE-INST-IN
                            (i (inst-at-stg '(LSU wbuf0 lch) MT)))
                 (:instance LSU-WBUF0-COMPLETE-IF-COMPLETE-INST-IN
                            (i (inst-at-stg '(complete wbuf0) MT)))
                 (:instance LSU-WBUF0-COMPLETE-IF-COMPLETE-INST-IN
                            (i (inst-at-stg '(commit wbuf0) MT))))
                :in-theory (disable UNIQ-INST-AT-LSU-WBUF0-IF-VALID
                                    LSU-WBUF0-COMPLETE-IF-COMPLETE-INST-IN)
                )))

(defthm uniq-inst-at-stg-LSU-wbuf1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (wbuf-complete? (LSU-wbuf1 (MA-LSU MA)))))
                  (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
             (uniq-inst-at-stg '(LSU wbuf1) MT))
  :hints (("goal" :use
                  ((:instance UNIQ-INST-AT-LSU-WBUF1-IF-VALID)
                   (:instance uniq-inst-at-stgs*
                              (stgs '((LSU WBUF1)
                                      (LSU WBUF1 LCH)
                                      (COMPLETE WBUF1)
                                      (COMMIT WBUF1))))
                   (:instance uniq-inst-at-stgs*
                              (stgs '((LSU WBUF1 LCH)
                                      (COMPLETE WBUF1)
                                      (COMMIT WBUF1))))
                   (:instance uniq-inst-at-stgs*
                              (stgs '((COMPLETE WBUF1)
                                      (COMMIT WBUF1))))
                   (:instance LSU-WBUF1-COMPLETE-IF-COMPLETE-INST-IN
                              (i (inst-at-stg '(LSU wbuf1 lch) MT)))
                   (:instance LSU-WBUF1-COMPLETE-IF-COMPLETE-INST-IN
                              (i (inst-at-stg '(complete wbuf1) MT)))
                   (:instance LSU-WBUF1-COMPLETE-IF-COMPLETE-INST-IN
                              (i (inst-at-stg '(commit wbuf1) MT))))
                  :in-theory (disable UNIQ-INST-AT-LSU-WBUF1-IF-VALID
                                      LSU-WBUF1-COMPLETE-IF-COMPLETE-INST-IN)
                  )))

(defthm uniq-inst-at-stg-commit-wbuf0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (wbuf-commit? (LSU-wbuf0 (MA-LSU MA))))
                  (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
             (uniq-inst-at-stg '(commit wbuf0) MT))
  :hints (("goal" :use
                  ((:instance UNIQ-INST-AT-LSU-WBUF0-IF-VALID)
                   (:instance uniq-inst-at-stgs*
                              (stgs '((LSU WBUF0)
                                      (LSU WBUF0 LCH)
                                      (COMPLETE WBUF0)
                                      (COMMIT WBUF0))))
                   (:instance uniq-inst-at-stgs*
                              (stgs '((LSU WBUF0 LCH)
                                      (COMPLETE WBUF0)
                                      (COMMIT WBUF0))))
                   (:instance uniq-inst-at-stgs*
```

```
                              (stgs '((COMPLETE WBUF0)
                                      (COMMIT WBUF0)))))
                  :in-theory (disable
                              UNIQ-INST-AT-LSU-WBUF0-IF-VALID))))

(defthm uniq-inst-at-stg-commit-wbuf1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (wbuf-commit? (LSU-wbuf1 (MA-LSU MA))))
                  (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
             (uniq-inst-at-stg '(commit wbuf1) MT))
  :hints (("goal" :use
                  ((:instance UNIQ-INST-AT-LSU-WBUF1-IF-VALID)
                   (:instance uniq-inst-at-stgs*
                              (stgs '((LSU WBUF1)
                                      (LSU WBUF1 LCH)
                                      (COMPLETE WBUF1)
                                      (COMMIT WBUF1))))
                   (:instance uniq-inst-at-stgs*
                              (stgs '((LSU WBUF1 LCH)
                                      (COMPLETE WBUF1)
                                      (COMMIT WBUF1))))
                   (:instance uniq-inst-at-stgs*
                              (stgs '((COMPLETE WBUF1)
                                      (COMMIT WBUF1)))))
                  :in-theory (disable
                              UNIQ-INST-AT-LSU-WBUF1-IF-VALID))))

(defthm not-INST-excpt-if-commit-stg
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (or (equal (INST-stg i) '(commit wbuf0))
                      (equal (INST-stg i) '(commit wbuf1)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (INST-excpt? I) 0))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

(local
(defthm not-no-inst-at-stgs-in-trace-if-member-equal
    (implies (and (member-equal i trace)
                  (member-equal (INST-stg i) stgs))
             (not (no-inst-at-stgs-in-trace stgs trace)))))

(local
(defthm not-no-inst-at-stg-in-trace-if-member-equal
    (implies (member-equal j trace)
             (not (no-inst-at-stg-in-trace (INST-stg j) trace)))))

(local
(defthm no-inst-at-stg-in-trace-if-no-inst-at-stgs-in-trace
    (implies (and (not (no-inst-at-stg-in-trace stg trace))
                  (member-equal stg stgs))
             (not (no-inst-at-stgs-in-trace stgs trace)))))

(encapsulate nil
(local
(defthm uniq-inst-at-LSU-lch-if-INST-in-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
```

```
                (uniq-inst-at-stgs-in-trace '((LSU LCH)
                                             (LSU WBUF0 LCH)
                                             (LSU WBUF1 LCH))
                                           trace)
                (equal (INST-stg i) '(LSU lch))
                (member-equal i trace) (INST-p i)
                (MAETT-p MT) (MA-state-p MA))
           (uniq-inst-at-stg-in-trace '(LSU lch) trace))))

(local
(defthm uniq-inst-at-LSU-lch-if-INST-in
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA)
                (equal (INST-stg i) '(LSU lch)))
           (uniq-inst-at-stg '(LSU lch) MT))
  :hints (("goal" :use ((:instance UNIQ-INST-AT-LSU-LCH-IF-VALID)
                        (:instance LSU-LCH-VALID-IF-INST-IN))
                :in-theory (e/d (uniq-inst-at-stg uniq-inst-at-stgs
                                                  equal-b1p-converter INST-in)
                                (UNIQ-INST-AT-LSU-LCH-IF-VALID))))))

(local
(defthm uniq-inst-at-LSU-wbuf0-lch-if-INST-in-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (uniq-inst-at-stgs-in-trace '((LSU LCH)
                                             (LSU WBUF0 LCH)
                                             (LSU WBUF1 LCH))
                                           trace)
                (equal (INST-stg i) '(LSU wbuf0 lch))
                (member-equal i trace) (INST-p i)
                (MAETT-p MT) (MA-state-p MA))
           (uniq-inst-at-stg-in-trace '(LSU wbuf0 lch) trace))))

(local
(defthm uniq-inst-at-LSU-wbuf0-lch-if-INST-in
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA)
                (equal (INST-stg i) '(LSU wbuf0 lch)))
           (uniq-inst-at-stg '(LSU wbuf0 lch) MT))
  :hints (("goal" :use ((:instance UNIQ-INST-AT-LSU-LCH-IF-VALID)
                        (:instance LSU-LCH-VALID-IF-INST-IN))
                :in-theory (e/d (uniq-inst-at-stg uniq-inst-at-stgs
                                                  equal-b1p-converter INST-in)
                                (UNIQ-INST-AT-LSU-LCH-IF-VALID))))))

(local
(defthm uniq-inst-at-LSU-wbuf1-lch-if-INST-in-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (uniq-inst-at-stgs-in-trace '((LSU LCH)
                                             (LSU WBUF0 LCH)
                                             (LSU WBUF1 LCH))
                                           trace)
                (equal (INST-stg i) '(LSU wbuf1 lch))
                (member-equal i trace) (INST-p i)
                (MAETT-p MT) (MA-state-p MA))
           (uniq-inst-at-stg-in-trace '(LSU wbuf1 lch) trace))))

(local
```

```
(defthm uniq-inst-at-LSU-wbuf1-lch-if-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU wbuf1 lch)))
             (uniq-inst-at-stg '(LSU wbuf1 lch) MT))
  :hints (("goal" :use ((:instance UNIQ-INST-AT-LSU-LCH-IF-VALID)
                        (:instance LSU-LCH-VALID-IF-INST-IN))
                  :in-theory (e/d (uniq-inst-at-stg uniq-inst-at-stgs
                                                    equal-b1p-converter INST-in)
                                  (UNIQ-INST-AT-LSU-LCH-IF-VALID))))))

; If i is at stage IFU, DQ or EXECUTE, then
; i is the uniq instruction at the stage (INST-stg i).
; This may not be true for other stages, because there can be multiple retired
; instructions in a MAETT.
(defthm uniq-inst-at-INST-stg-if-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (or (IFU-stg-p (INST-stg i))
                      (DQ-stg-p (INST-stg i))
                      (execute-stg-p (INST-stg i))))
             (uniq-inst-at-stg (INST-stg i) MT))
  :hints (("goal" :in-theory (enable IFU-stg-p DQ-stg-p execute-stg-p
                                     IU-stg-p MU-stg-p BU-stg-p
                                     LSU-stg-p))))
)

(encapsulate nil
(local
(defthm uniq-stage-inst-induct
    (implies (and (equal (INST-stg i) (INST-stg j))
                  (member-equal i trace)
                  (member-equal j trace)
                  (not (equal i j)))
             (not (uniq-inst-at-stg-in-trace (INST-stg i) trace)))))

(local
(defthm uniq-stage-inst-help
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg (INST-stg i) MT)
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) (INST-stg j)))
             (equal i j))
  :hints (("goal" :in-theory (enable INST-in uniq-inst-at-stg)))
  :rule-classes nil))


; A corollary of uniq-inst-at-INST-stg-if-INST-in.
(defthm uniq-stage-inst
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) (INST-stg j))
                  (or (IFU-stg-p (INST-stg i))
                      (DQ-stg-p (INST-stg i))
                      (execute-stg-p (INST-stg i))))
             (equal i j))
```

```
  :hints (("goal" :use (:instance uniq-stage-inst-help)))
  :rule-classes nil)
)

(in-theory (disable uniq-inst-at-INST-stg-if-INST-in))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;; ROB Related Lemmas Index
; Lemmas about Uniq-INST and robe-valid bits.
; Stages of inst-of-tag
; INST-inv of inst-of-tag
; ROB-head(MA-rob)=MT-rob-head(MT) and similar lemmas
; Lemmas about robe-complete? fields
; Lemmas about consistent-robe-p
; Lemmas about robe identity
; Lemmas related to rob-empty? rob-full?
; Lemmas about other fields.
; Lemmas to relate inst-of-tag and INST-at-stg
; definition and lemmas about tag-in-order

;  Related lemmas:
;   not-commit-jmp-if-rob-head-complete-wbuf
;   not-leave-excpt-if-rob-head-complete-wbuf
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defthm rob-index-p-minis-8
    (implies (and (integerp idx) (< 0 idx) (<= idx *rob-size*))
             (rob-index-p (- *rob-size* idx)))
  :hints (("goal" :in-theory (enable rob-index-p unsigned-byte-p))))

(defthm rob-index-p-minis-x-y
    (implies (and (integerp x) (integerp y)  (<= 0 y) (<= y x)
                  (< x *rob-size*))
             (rob-index-p (- x y)))
  :hints (("goal" :in-theory (enable rob-index-p unsigned-byte-p))))

(defthm len-ROB-entries
    (implies (ROB-p ROB)
             (equal (len (ROB-entries ROB)) *rob-size*))
  :hints (("goal" :in-theory (enable ROB-p ROB-ENTRIES-P))))

(defthm INST-tag-ge-0
    (implies (INST-p i) (<= 0 (INST-tag i)))
  :rule-classes :linear)

(defthm INST-tag-lt-8
    (implies (INST-p i) (< (INST-tag i) 8))
  :rule-classes :linear)

;;;;;;Begin of lemmas about Uniq-INST predicates and valid bits;;;;;;
;; inst-of-tag is an INST-p if there is an instruction with the tag.
(encapsulate nil
(local
(defthm INST-p-inst-of-tag-in-trace
    (implies (and (uniq-inst-of-tag-in-trace stg trace)
                  (INST-listp trace))
             (INST-p (inst-of-tag-in-trace stg trace)))))

(defthm INST-p-inst-of-tag
    (implies (and (uniq-inst-of-tag rix MT)
                  (MAETT-p MT))
             (INST-p (inst-of-tag rix MT)))
  :hints (("Goal" :in-theory (enable uniq-inst-of-tag inst-of-tag))))
```

```
)

(local
(defthm no-inst-of-tag-in-trace-INST-tag-if-member
    (implies (and (member-equal i trace)
                  (dispatched-p i)
                  (not (committed-p i)))
             (not (no-inst-of-tag-in-trace (INST-tag i) trace)))
  :hints (("goal" :in-theory (enable dispatched-p committed-p)))))

(local
(defthm member-equal-inst-of-tag-in-trace
    (implies (uniq-inst-of-tag-in-trace rix trace)
             (member-equal (inst-of-tag-in-trace rix trace) trace))))

;;; inst-of-tag belongs to a MAETT.
(defthm INST-in-inst-of-tag
    (implies (uniq-inst-of-tag rix MT)
             (INST-in (inst-of-tag rix MT) MT))
  :Hints (("Goal" :in-theory (enable inst-of-tag uniq-inst-of-tag
                                     INST-in))))

(encapsulate nil
(local
(defthm uniq-inst-of-tag-no-inst-of-tag-exclusive-help
    (implies (uniq-inst-of-tag-in-trace rix trace)
             (not (no-inst-of-tag-in-trace rix trace)))))

(defthm uniq-inst-of-tag-no-inst-of-tag-exclusive
    (implies (uniq-inst-of-tag rix MT)
             (not (no-inst-of-tag rix MT)))
  :hints (("goal" :in-theory (enable no-inst-of-tag uniq-inst-of-tag)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (no-inst-of-tag rix MT)
                      (not (uniq-inst-of-tag rix MT))))))
)

(encapsulate nil
(local
(defthm uniq-inst-of-tag-in-trace-if-robe-valid-help-help-help
    (implies (and (no-tag-conflict-at rix MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (b1p (robe-valid? (nth-robe rix (MA-rob MA)))))
             (uniq-inst-of-tag rix MT))
  :hints (("Goal" :in-theory (enable no-tag-conflict-at)))))

(local
(defthm uniq-inst-of-tag-in-trace-if-robe-valid-help-help
    (implies (and (no-tag-conflict-under upper MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (integerp rix)
                  (<= 0 rix)
                  (integerp upper)
                  (< rix upper)
                  (b1p (robe-valid? (nth-robe rix (MA-rob MA)))))
             (uniq-inst-of-tag rix MT))))

(local
```

```
(defthm uniq-inst-of-tag-in-trace-if-robe-valid-help
    (implies (and (no-tag-conflict MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (rob-index-p rix)
                  (b1p (robe-valid? (nth-robe rix (MA-rob MA)))))
             (uniq-inst-of-tag rix MT))
  :hints (("Goal" :in-theory (enable rob-index-p unsigned-byte-p
                                     no-tag-conflict)))))

; relation between robe-valid? and uniq-inst-of-tag.
(defthm uniq-inst-of-tag-if-robe-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (rob-index-p rix)
                  (b1p (robe-valid? (nth-robe rix (MA-rob MA)))))
             (uniq-inst-of-tag rix MT))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (MAETT-p MT)
                           (MA-state-p MA)
                           (rob-index-p rix)
                           (b1p (robe-valid? (nth-robe rix (MA-rob MA)))))
                      (uniq-inst-of-tag-in-trace rix (MT-trace MT)))
             :hints (("goal" :in-theory (enable uniq-inst-of-tag))))))

) ; encapsulate

(encapsulate nil
(local
(defthm no-inst-of-tag-in-trace-if-not-robe-valid-help-help-help
    (implies (and (no-tag-conflict-at rix MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (not (b1p (robe-valid? (nth-robe rix (MA-rob MA))))))
             (no-inst-of-tag rix MT))
  :hints (("Goal" :in-theory (enable no-tag-conflict-at)))))

(local
(defthm no-inst-of-tag-in-trace-if-not-robe-valid-help-help
    (implies (and (no-tag-conflict-under upper MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (integerp rix)
                  (<= 0 rix)
                  (integerp upper)
                  (< rix upper)
                  (not (b1p (robe-valid? (nth-robe rix (MA-rob MA))))))
             (no-inst-of-tag rix MT))))

(local
(defthm no-inst-of-tag-in-trace-if-not-robe-valid-help
    (implies (and (no-tag-conflict MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (rob-index-p rix)
                  (not (b1p (robe-valid? (nth-robe rix (MA-rob MA))))))
             (no-inst-of-tag rix MT))
  :hints (("Goal" :in-theory (enable rob-index-p unsigned-byte-p
```

```
                                    no-tag-conflict)))))

(defthm no-inst-of-tag-if-not-robe-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (rob-index-p rix)
                  (not (b1p (robe-valid? (nth-robe rix (MA-rob MA))))))
             (no-inst-of-tag rix MT)))
)

(encapsulate nil
(local
(defthm robe-valid-nth-robe-INST-tag-help
    (implies (and (INST-inv i MA)
                  (MA-state-p MA)
                  (INST-p i)
                  (dispatched-p i)
                  (not (committed-p i)))
             (b1p (robe-valid? (nth-robe (INST-tag i) (MA-rob MA)))))
  :hints (("Goal" :in-theory (enable inst-inv-def
                                     lift-b-ops)))))

(defthm robe-valid-nth-robe-INST-tag
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (INST-p i)
                  (dispatched-p i)
                  (not (committed-p i)))
             (b1p (robe-valid? (nth-robe (INST-tag i) (MA-rob MA)))))
  :rule-classes
  ((:rewrite :corollary
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (INST-p i)
                  (dispatched-p i)
                  (not (committed-p i)))
             (equal (robe-valid? (nth-robe (INST-tag i) (MA-rob MA))) 1))
    :hints (("goal" :in-theory (enable equal-b1p-converter lift-b-ops))))))
)

(encapsulate nil
(local
(defthm uniq-inst-of-tag-tag-if-INST-in-help
    (implies (and (no-tag-conflict-at (INST-tag i) MT MA)
                  (INST-inv i MA)
                  (MAETT-p MT)
                  (INST-p i)
                  (MA-state-p MA)
                  (dispatched-p i)
                  (not (committed-p i)))
             (uniq-inst-of-tag (INST-tag i) MT))
  :hints (("goal" :In-theory (enable no-tag-conflict-at
                                     inst-inv-def)))
  :rule-classes nil))

(defthm uniq-inst-of-tag-INST-tag-if-INST-in
    (implies (and (inv MT MA)
```

```
                (MAETT-p MT)
                (MA-state-p MA)
                (INST-p i)
                (INST-in i MT)
                (dispatched-p i)
                (not (committed-p i)))
           (uniq-inst-of-tag (INST-tag i) MT))
  :hints (("goal" :use (:instance uniq-inst-of-tag-tag-if-INST-in-help))))
)

;; Following two lemmas show that INST-tag and inst-of-tag
;; are inverse functions of each other in a sense.

(encapsulate nil
(local
(defthm INST-tag-inst-of-tag-in-trace
    (implies (and (INST-listp trace)
                (rob-index-p rix)
                (uniq-inst-of-tag-in-trace rix trace))
           (equal (INST-tag (inst-of-tag-in-trace rix trace)) rix))))

(defthm INST-tag-inst-of-tag
    (implies (and (MAETT-p MT)
                (rob-index-p rix)
                (uniq-inst-of-tag rix MT))
           (equal (INST-tag (inst-of-tag rix MT)) rix))
  :hints (("Goal" :in-theory (enable inst-of-tag uniq-inst-of-tag))))
)

(encapsulate nil
(local
(defthm inst-of-tag-in-trace-INST-tag
    (implies (and (INST-listp trace)
                (INST-p i)
                (member-equal i trace)
                (uniq-inst-of-tag-in-trace (INST-tag i) trace)
                (dispatched-p i)
                (not (committed-p i)))
           (equal (inst-of-tag-in-trace (INST-tag i) trace) i))
  :hints (("goal" :in-theory (enable dispatched-p committed-p)))))

(local
(defthm inst-of-tag-INST-tag-help
    (implies (and (inv MT MA)
                (MAETT-p MT)
                (INST-p i)
                (INST-in i MT)
                (uniq-inst-of-tag (INST-tag i) MT)
                (dispatched-p i)
                (not (committed-p i)))
           (equal (inst-of-tag (INST-tag i) MT) i))
  :hints (("Goal" :in-theory (enable inst-of-tag INST-in uniq-inst-of-tag)))))

(defthm inst-of-tag-INST-tag
    (implies (and (inv MT MA)
                (MAETT-p MT)
                (INST-p i)
                (MA-state-p MA)
                (INST-in i MT)
                (dispatched-p i)
                (not (committed-p i)))
           (equal (inst-of-tag (INST-tag i) MT) i)))
```

```
)

(encapsulate nil
(local
 (defthm not-no-inst-of-tag-INST-stg-if-member-equal
     (implies (and (member-equal i trace)
                   (dispatched-p i ) (not (committed-p i)))
              (not (no-inst-of-tag-in-trace (INST-tag i) trace)))
   :hints (("goal" :in-theory (enable dispatched-p committed-p)))))

(defthm not-no-inst-of-tag-if-inst-in
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (dispatched-p i) (not (committed-p i)))
             (not (no-inst-of-tag (INST-tag i) MT)))
  :hints (("goal" :in-theory (enable inst-in no-inst-of-tag))))
)

;;;;;;End of lemmas about Uniq-INST predicates and valid bits;;;;;;

;;;;; Begin of lemmas about stages of inst-of-tag ;;;;;
(encapsulate nil
(local
(defthm execute-or-complete-stg-p-inst-of-tag-in-trace
  (implies (and (not (execute-stg-p
                       (INST-stg (inst-of-tag-in-trace rix trc))))
                (not (complete-stg-p
                       (INST-stg (inst-of-tag-in-trace rix trc)))))
           (not (uniq-inst-of-tag-in-trace rix trc)))
  :hints (("goal" :in-theory (enable committed-p dispatched-p)))))

(defthm execute-or-complete-stg-p-inst-of-tag
    (implies (uniq-inst-of-tag rix MT)
             (and (dispatched-p (inst-of-tag rix MT))
                  (not (committed-p (inst-of-tag rix MT)))))
  :rule-classes
  ((:rewrite :corollary
             (implies (not (dispatched-p (inst-of-tag rix MT)))
                      (not (uniq-inst-of-tag rix MT))))
   (:rewrite :corollary
             (implies (committed-p (inst-of-tag rix MT))
                      (not (uniq-inst-of-tag rix MT))))
   (:rewrite :corollary
     (implies (and (not (execute-stg-p (INST-stg (inst-of-tag rix MT))))
                   (not (complete-stg-p (INST-stg (inst-of-tag rix MT)))))
              (not (uniq-inst-of-tag rix MT)))
     :hints (("goal" :in-theory (e/d (committed-p dispatched-p)
                                     (INST-OF-TAG-IS-DISPATCHED
                                      INST-OF-TAG-IS-NOT-COMMITTED))))))
  :hints (("Goal" :in-theory (enable inst-of-tag uniq-inst-of-tag
                                     dispatched-p committed-p))))
) ; encapsulate
;;;;; End of lemmas about stages of inst-of-tag ;;;;;

;;;;; INST-inv about inst-of-tag
(defthm INST-inv-inst-of-tag
    (implies (and (inv MT MA)
                  (uniq-inst-of-tag rix MT))
             (INST-inv (inst-of-tag rix MT) MA)))

;;;;; rob-head-MA-rob-=-MT-rob-head and similar lemmas.
```

```
;; Rob-head field of MA and ROB-head field of MAETT are equal.
(defthm rob-head-MA-rob-=-MT-rob-head
    (implies (and (inv MT MA)
                  (MA-state-p MA)
                  (MAETT-p MT))
             (equal (rob-head (MA-rob MA)) (MT-rob-head MT)))
  :hints (("goal" :in-theory (enable weak-inv inv
                                     misc-inv))))

;; Rob-tail field of MA and ROB-tail field of MAETT are equal.
(defthm rob-tail-MA-rob-=-MT-rob-tail
    (implies (and (inv MT MA)
                  (MA-state-p MA)
                  (MAETT-p MT))
             (equal (rob-tail (MA-rob MA)) (MT-rob-tail MT)))
  :hints (("goal" :in-theory (enable weak-inv inv
                                     misc-inv))))

(defthm MA-rob-flg-MT-rob-flg
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (rob-flg (MA-rob MA)) (MT-rob-flg MT)))
  :hints (("Goal" :in-theory (enable inv misc-inv))))

(defthm MT-ROB-tail-le-MT-rob-head
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (ROB-flg (MA-ROB MA))))
             (<= (MT-ROB-tail MT) (MT-ROB-head MT)))
  :hints (("Goal" :in-theory (e/d (consistent-ROB-flg-p
                                   consistent-MA-p consistent-rob-p
                                   lift-b-ops)
                                  (consistent-rob-flg-p-forward))
                  :cases ((consistent-MA-p MA))))
  :rule-classes :linear)

(defthm MT-ROB-tail-ge-MT-rob-head
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (ROB-flg (MA-ROB MA)))))
             (>= (MT-ROB-tail MT) (MT-ROB-head MT)))
  :hints (("Goal" :in-theory (e/d (consistent-ROB-flg-p
                                   consistent-MA-p consistent-rob-p
                                   lift-b-ops)
                                  (consistent-rob-flg-p-forward))
                  :cases ((consistent-MA-p MA))))
  :rule-classes :linear)

(defthm MT-ROB-head-ge-0
    (implies (MAETT-p MT) (>= (MT-rob-head MT) 0))
  :rule-classes :linear)

(defthm MT-ROB-tail-ge-0
    (implies (MAETT-p MT) (>= (MT-rob-tail MT) 0))
  :rule-classes :linear)

(defthm MT-ROB-head-lt-8
    (implies (MAETT-p MT) (< (MT-rob-head MT) 8))
  :rule-classes :linear)

(defthm MT-ROB-tail-lt-8
    (implies (MAETT-p MT) (< (MT-rob-tail MT) 8))
```

```
  :rule-classes :linear)

(defthm INST-in-order-inst-of-tag-not-dispatched
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (uniq-inst-of-tag rix MT)
                  (not (dispatched-p i)))
             (INST-in-order-p (inst-of-tag rix MT) i MT)))

(encapsulate nil
; If there is a dispatched but not yet committed instruction, the ROB
; is not empty.  Thus there is an instruction at MT-head
(local
(defthm not-uniq-inst-of-tag-in-trace-if-no-dispatched-inst-p
    (implies (no-dispatched-inst-p trace)
             (not (uniq-inst-of-tag-in-trace rix trace)))))

(local
(defthm uncommitted-inst-p-is-after-MT-ROB-head-help
    (implies (and (INST-listp trace)
                  (distinct-member-p trace)
                  (in-order-trace-p trace)
                  (in-order-rob-trace-p trace rob-head)
                  (uniq-inst-of-tag-in-trace rob-head trace)
                  (INST-p i) (member-equal i trace)
                  (not (committed-p i))
                  (not (equal (inst-of-tag-in-trace rob-head trace)
                              i)))
             (member-in-order (inst-of-tag-in-trace rob-head trace) i trace))
  :hints (("goal" :in-theory (enable committed-p dispatched-p
                                     member-in-order*)))))

(defthm uncommitted-inst-p-is-after-MT-ROB-head
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-of-tag (MT-rob-head MT) MT)
                  (INST-p i) (INST-in i MT)
                  (not (committed-p i))
                  (not (equal (inst-of-tag (MT-rob-head MT) MT) i)))
             (INST-in-order-p (inst-of-tag (MT-rob-head MT) MT) i MT))
  :hints (("goal" :in-theory (enable inv in-order-dispatch-commit-p
                                     weak-inv
                                     MT-distinct-INST-p
                                     in-order-rob-p
                                     INST-in-order-p inst-of-tag
                                     uniq-inst-of-tag INST-in))))
)

(defthm INST-tag-step-INST-at-DQ
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg '(DQ 0) MT)
                  (b1p (dispatch-inst? MA)))
             (equal (INST-tag (step-INST (INST-at-stg '(DQ 0) MT)
                                         MT MA sigs))
                    (MT-rob-tail MT)))
  :hints (("Goal" :in-theory (enable step-inst-DQ-inst
                                     step-inst-low-level-functions))))
;;;;;;;;;Lemmas about robe-complete? fields;;;;;;;;;;;;;;;;;;
(encapsulate nil
(local
```

```
(defthm robe-execute-nth-robe-INST-tag-help
    (implies (and (INST-inv i MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (INST-p i)
                  (execute-stg-p (INST-stg i)))
             (equal (robe-complete? (nth-robe (INST-tag i) (MA-rob MA)))
                    0))
  :hints (("Goal" :in-theory (enable inst-inv-def
                                     equal-b1p-converter
                                     lift-b-ops)))))

(defthm robe-execute-nth-robe-INST-tag
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (INST-p i)
                  (INST-in i MT)
                  (execute-stg-p (INST-stg i)))
             (equal (robe-complete? (nth-robe (INST-tag i) (MA-rob MA))) 0)))
)

(encapsulate nil
(local
(defthm robe-execute-nth-robe-rix-help
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (rob-index-p rix)
                  (uniq-inst-of-tag rix MT)
                  (execute-stg-p (INST-stg (inst-of-tag rix MT))))
             (equal (robe-complete? (nth-robe (INST-tag (inst-of-tag rix MT))
                                              (MA-rob MA)))
                    0))
  :hints (("goal" :in-theory (disable  INST-TAG-INST-OF-TAG)))
  :rule-classes nil)
)

(defthm robe-execute-nth-robe-rix
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (rob-index-p rix)
                  (uniq-inst-of-tag rix MT)
                  (execute-stg-p (INST-stg (inst-of-tag rix MT))))
             (equal (robe-complete? (nth-robe rix (MA-rob MA))) 0))
  :hints (("Goal" :use (:instance robe-execute-nth-robe-rix-help))))
)

(encapsulate nil
(local
(defthm robe-complete-nth-robe-INST-tag-help
    (implies (and (INST-inv i MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (INST-p i)
                  (complete-stg-p (INST-stg i)))
             (equal (robe-complete? (nth-robe (INST-tag i) (MA-rob MA))) 1))
  :hints (("Goal" :in-theory (enable inst-inv-def
                                     equal-b1p-converter
                                     lift-b-ops)))))
```

```
(defthm robe-complete-nth-robe-INST-tag
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (INST-p i)
                  (INST-in i MT)
                  (complete-stg-p (INST-stg i)))
             (equal (robe-complete? (nth-robe (INST-tag i) (MA-rob MA))) 1)))
) ;encapsulate

(encapsulate nil
(local
(defthm robe-complete-nth-robe-rix-help
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (rob-index-p rix)
                  (uniq-inst-of-tag rix MT)
                  (complete-stg-p (INST-stg (inst-of-tag rix MT))))
             (equal (robe-complete? (nth-robe (INST-tag (inst-of-tag rix MT))
                                              (MA-rob MA)))
                    1))
  :hints (("goal" :in-theory (disable  INST-TAG-INST-OF-TAG)))
  :rule-classes nil)
)

(defthm robe-complete-nth-robe-rix
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (rob-index-p rix)
                  (uniq-inst-of-tag rix MT)
                  (complete-stg-p (INST-stg (inst-of-tag rix MT))))
             (equal (robe-complete? (nth-robe rix (MA-rob MA))) 1))
  :hints (("Goal" :use (:instance robe-complete-nth-robe-rix-help))))
)

(defthm complete-stg-if-robe-complete
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (rob-index-p rix)
                  (uniq-inst-of-tag rix MT)
                  (b1p (robe-complete? (nth-robe rix (MA-rob MA)))))
             (complete-stg-p (INST-stg (inst-of-tag rix MT))))
  :hints (("goal" :use (:instance INST-is-at-one-of-the-stages
                                  (i (inst-of-tag rix MT))))))

(defthm execute-stg-if-not-robe-complete
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (rob-index-p rix)
                  (uniq-inst-of-tag rix MT)
                  (not (b1p (robe-complete? (nth-robe rix (MA-rob MA))))))
             (execute-stg-p (INST-stg (inst-of-tag rix MT))))
  :hints (("goal" :use (:instance INST-is-at-one-of-the-stages
                                  (i (inst-of-tag rix MT))))))

(in-theory (disable complete-stg-if-robe-complete
                    execute-stg-if-not-robe-complete))
```

```
(defthm robe-complete-if-complete-stg-inst-of-tag
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (rob-index-p rix)
                  (uniq-inst-of-tag rix MT))
             (equal (b1p (robe-complete? (nth-robe rix (MA-rob MA))))
                    (complete-stg-p (INST-stg (inst-of-tag rix MT)))))
  :hints (("goal" :use (:instance INST-is-at-one-of-the-stages
                                  (i (inst-of-tag rix MT))))))

(in-theory (disable robe-complete-if-complete-stg-inst-of-tag))

;;;;; End of lemmas about robe-complete? field

;;;;; Lemmas about consistent-robe-p
(encapsulate nil
(local
(defthm rob-index-+-1
    (implies (rob-index-p idx)
             (equal (rob-index (+ 1 idx))
                    (b-if (logbit *ROB-INDEX-SIZE* (+ 1 idx))
                          0 (+ 1 idx))))
  :hints (("goal" :in-theory (enable lift-b-ops rob-index-p rob-index
                                     unsigned-byte-p)
                  :cases ((B1P (LOGBIT 3 (+ 1 IDX)))))
          ("subgoal 1" :use (:instance logbit-0-if-val-lt-expt-2-width
                                       (val (+ 1 idx)) (width 3))))))

(local
(defthm logbit3-0-if-rob-index-not-7
    (implies (rob-index-p rix)
             (iff (b1p (logbit 3 (+ 1 rix))) (equal rix 7)))
  :Hints (("goal" :use (:instance logbit-0-if-val-lt-expt-2-width
                                  (val (+ 1 rix)) (width 3))
                  :in-theory (enable rob-p rob-index-p unsigned-byte-p)))))


(local
 (defthm consistent-robe-p-nth-robe-help
     (implies (and (ROBE-listp entries)
                   (equal (len entries) (- *ROB-SIZE* idx))
                   (rob-index-p idx)
                   (rob-index-p rix)
                   (<= idx rix)
                   (consistent-rob-entries-p entries idx ROB))
              (consistent-robe-p (nth (- rix idx) entries) rix rob))
   :hints (("goal" :in-theory (enable rob-index-p unsigned-byte-p)))
   :rule-classes nil))

; Any rob-entry satisfies consistent-robe-p.
; The proof requires logbit-0-if-val-lt-expt-2-width
(defthm consistent-robe-p-nth-robe
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (rob-index-p rix))
             (consistent-robe-p (nth-robe rix (MA-ROB MA))
                                rix (MA-rob MA)))
  :hints (("goal" :in-theory (enable inv consistent-rob-p
                                     nth-robe
                                     rob-index-p unsigned-byte-p
                                     consistent-MA-p)
```

```
                :use (:instance consistent-robe-p-nth-robe-help
                                (idx 0)
                                (rob (MA-rob MA))
                                (entries (ROB-entries (MA-rob MA)))))))
)
;;;;; End of Lemmas about consistent-robe-p
;;;;; Lemmas about tag identity
(encapsulate nil
(local
(defthm not-member-equal-if-no-inst-of-tag-in-trace
    (implies (and (no-inst-of-tag-in-trace (INST-tag j) trace)
                  (dispatched-p j) (not (committed-p j)))
             (not (member-equal j trace)))
  :hints (("goal" :in-theory (enable committed-p dispatched-p)))))

(local
(defthm tag-identity-help3
    (implies (and (uniq-inst-of-tag-in-trace rix trace)
                  (INST-p i) (member-equal i trace)
                  (dispatched-p i) (not (committed-p i))
                  (INST-p j) (member-equal j trace)
                  (dispatched-p j) (not (committed-p j))
                  (equal (INST-tag i) rix)
                  (equal (INST-tag j) rix))
             (equal i j))
  :hints (("goal" :in-theory (enable dispatched-p committed-p)
                  :induct t))
  :rule-classes nil))

(local
(defthm tag-identity-help-help
    (implies (and (uniq-inst-of-tag (INST-tag i) MT)
                  (INST-p i) (INST-in i MT)
                  (dispatched-p i) (not (committed-p i))
                  (INST-p j) (INST-in j MT)
                  (dispatched-p j) (not (committed-p j)))
             (equal (equal (INST-tag i) (INST-tag j))
                    (equal i j)))
  :hints (("goal" :use (:instance tag-identity-help3
                                  (rix (INST-tag i)) (trace (MT-trace MT)))
                  :in-theory (enable uniq-inst-of-tag
                                     INST-in)))))

(local
(defthm tag-identity-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-tag-conflict-at (INST-tag i) MT MA)
                  (INST-p i) (INST-in i MT)
                  (dispatched-p i) (not (committed-p i))
                  (INST-p j) (INST-in j MT)
                  (dispatched-p j) (not (committed-p j)))
             (equal (equal (INST-tag i) (INST-tag j))
                    (equal i j)))
  :hints (("goal" :in-theory (e/d (no-tag-conflict-at)
                                  (NO-TAG-CONFLICT-AT-ALL-RIX
                                   UNIQ-INST-OF-TAG-INST-TAG-IF-INST-IN
                                   UNIQ-INST-OF-TAG-IF-ROBE-VALID))))))

(defthm tag-identity
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
```

```
                (INST-p i) (INST-in i MT)
                (dispatched-p i) (not (committed-p i))
                (INST-p j) (INST-in j MT)
                (dispatched-p j) (not (committed-p j)))
           (equal (equal (INST-tag i) (INST-tag j))
                  (equal i j)))
  :hints (("goal" :restrict ((tag-identity-help ((MA MA) (MT MT)))))))
)

;;;;; Begin of Lemmas related to rob-empty?
(encapsulate nil
(local
(defthm zbp-robe-valid-if-rob-empty-help-help
    (implies (AND (ROB-P ROB)
                (B1P (ROB-EMPTY? ROB))
                (B1P (ROBE-VALID? robe)))
           (not (CONSISTENT-ROBE-P robe rix ROB)))
  :hints (("goal" :in-theory (enable consistent-robe-p rob-empty?
                                     lift-b-ops bv-eqv-iff-equal)))))

(local
(defthm zbp-robe-valid-if-rob-empty-help
    (IMPLIES (AND (B1P (ROBE-VALID? (NTH RIX entries)))
                (ROB-P ROB)
                (B1P (ROB-EMPTY? ROB))
                (robe-listp entries)
                (< rix (len entries)))
           (not (CONSISTENT-ROB-ENTRIES-P entries rix2 ROB)))))

(defthm not-robe-valid-if-rob-empty-and-consistent-rob
    (implies (and (ROB-p rob)
                (rob-index-p rix)
                (b1p (rob-empty? rob))
                (consistent-rob-p rob))
           (equal (robe-valid? (nth-robe rix rob)) 0))
  :hints (("goal" :in-theory (enable nth-robe rob-entries-p rob-p
                                     equal-b1p-converter
                                     lift-b-ops
                                     consistent-rob-p))))
)

(defthm not-robe-valid-nth-robe-if-rob-empty
    (implies (and (inv MT MA)
                (MA-state-p MA)
                (rob-index-p rix)
                (b1p (rob-empty? (MA-rob MA))))
           (equal (robe-valid? (nth-robe rix (MA-rob MA))) 0))
  :hints (("goal" :in-theory (enable consistent-rob-p-forward))))

(defthm robe-not-head-if-INST-at-execute-stg-while-commit
    (implies (and (inv MT MA)
                (INST-p i)
                (MAETT-p MT)
                (MA-state-p MA)
                (INST-in i MT)
                (b1p (commit-inst? MA))
                (execute-stg-p (INST-stg i)))
           (not (equal (INST-tag i) (MT-rob-head MT))))
  :hints  (("goal" :in-theory (e/d (commit-inst? lift-b-ops)
                                   ()))))

(defthm not-rob-empty-if-INST-is-executed
```

```
    (implies (and (inv MT MA)
                  (dispatched-p i)
                  (not (committed-p i))
                  (INST-in i MT)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (INST-p i))
             (equal (rob-empty? (MA-rob MA)) 0))
  :hints (("Goal" :use (:instance not-robe-valid-nth-robe-if-rob-empty
                                  (rix (INST-tag i)))
                  :in-theory
                  (e/d (equal-b1p-converter lift-b-ops)
                       (not-robe-valid-nth-robe-if-rob-empty
                        NOT-ROBE-VALID-IF-ROB-EMPTY-AND-CONSISTENT-ROB))))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (execute-stg-p (INST-stg i))
                           (INST-in i MT)
                           (MAETT-p MT)
                           (MA-state-p MA)
                           (INST-p i))
                      (equal (rob-empty? (MA-rob MA)) 0)))
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (complete-stg-p (INST-stg i))
                           (INST-in i MT)
                           (MAETT-p MT)
                           (MA-state-p MA)
                           (INST-p i))
                      (equal (rob-empty? (MA-rob MA)) 0)))))

(encapsulate nil
(local
(defthm robe-empty-if-MT-rob-head-tail-matches-help
    (implies (and (consistent-rob-entries-p
                   (nthcdr (- idx n) (ROB-entries ROB)) (rob-index (- idx n))
                   ROB)
                  (integerp idx) (integerp n)
                  (<= 0 n) (<= n idx) (< idx *rob-size*)
                  (ROB-p rob)
                  (equal (rob-tail ROB) (rob-head ROB))
                  (not (b1p (rob-flg ROB))))
             (equal (robe-empty? idx ROB) 1))
  :Hints (("Goal" :in-theory (enable consistent-rob-entries-p
                                     ROBE-EMPTY? nth-robe lift-b-ops
                                     equal-b1p-converter
                                     consistent-robe-p)
                  :expand (CONSISTENT-ROB-ENTRIES-P
                           (NTHCDR (+ IDX (- N)) (ROB-ENTRIES ROB))
                           (ROB-INDEX (+ IDX (- N)))
                           ROB)
                  :induct (natural-induction n)))
  :rule-classes nil))

(defthm robe-empty-if-MT-rob-head-tail-matches
    (implies (and (inv MT MA)
                  (equal (MT-rob-tail MT) (MT-rob-head MT))
                  (rob-index-p idx)
                  (not (b1p (MT-rob-flg MT)))
                  (MAETT-p MT) (MA-state-p MA))
```

```
            (equal (robe-empty? idx (MA-ROB MA)) 1))
  :Hints (("Goal" :in-theory (e/d (rob-index-p CONSISTENT-MA-P
                                               UNSIGNED-BYTE-P
                                               consistent-rob-p)
                                  (CONSISTENT-MA-P-forward))
                  :use ((:instance robe-empty-if-MT-rob-head-tail-matches-help
                                   (n idx)
                                   (ROB (MA-rob MA)))
                        (:instance consistent-MA-p-forward
                                   (MA MA))))))
)

(defthm robe-empty-under-if-head-tail-matches
    (implies (and (inv MT MA)
                  (equal (MT-rob-tail MT) (MT-rob-head MT))
                  (not (b1p (MT-rob-flg MT)))
                  (integerp idx) (<= 0 idx) (<= idx *rob-size*)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (robe-empty-under? idx (MA-ROB MA)) 1))
  :hints (("goal" :in-theory (enable lift-b-ops ROBE-EMPTY-UNDER?
                                     rob-index-p unsigned-byte-p
                                     equal-b1p-converter)
                  :induct (natural-induction idx))))

(encapsulate nil
(local
(defthm ROB-valid-MT-rob-head-if-not-ROB-empty-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (ROB-entry-p robe)
                  (consistent-robe-p robe (MT-ROB-head MT) (MA-ROB MA))
                  (consistent-rob-flg-p (MA-ROB MA))
                  (not (b1p (ROB-empty? (MA-rob MA)))))
             (equal (ROBE-valid? robe) 1))
  :hints (("goal" :in-theory (enable consistent-robe-p
                                     consistent-rob-flg-p
                                     ROB-empty? lift-b-ops
                                     equal-b1p-converter)))))

; ROB-empty? is not on, MT-ROB-head contains a valid instruction.
(defthm ROB-valid-MT-rob-head-if-not-ROB-empty
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (ROB-empty? (MA-rob MA)))))
             (equal (ROBE-valid? (nth-robe (MT-ROB-head MT) (MA-ROB MA))) 1))
  :hints (("goal" :in-theory (enable consistent-ROB-p-forward
                                     consistent-ROB-flg-p-forward))))
)

(defthm not-commit-inst-if-rob-empty
    (implies (and  (inv MT MA)
                   (b1p (rob-empty? (MA-ROB MA)))
                   (MAETT-p MT) (MA-state-p MA))
             (equal (commit-inst? MA) 0))
  :hints (("Goal" :in-theory (enable commit-inst?))))

; If ROB is not empty, ROB contains an instruction at index MT-ROB-head.
(defthm uniq-inst-of-tag-MT-rob-head-if-not-empty
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (ROB-empty? (MA-rob MA)))))
             (uniq-inst-of-tag (MT-rob-head MT) MT)))
```

```
(defthm MT-rob-head-MT-rob-tail-if-rob-empty
    (implies (and (inv MT MA)
                  (b1p (rob-empty? (MA-rob MA)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (MT-rob-tail MT) (MT-ROB-head MT)))
  :hints (("Goal" :in-theory (enable rob-empty? lift-b-ops))))

; When ROB is not full, the entry pointed by ROB-tail is free.
(defthm robe-valid-MT-ROB-tail-if-not-rob-full
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (rob-full? (MA-ROB MA)))))
             (equal (robe-valid? (nth-robe (MT-ROB-tail MT) (MA-rob MA)))
                    0))
  :hints (("goal" :in-theory (e/d (rob-full? lift-b-ops
                                             consistent-robe-p)
                                  (CONSISTENT-ROBE-P-NTH-ROBE))
                  :use (:instance CONSISTENT-ROBE-P-NTH-ROBE
                                  (rix (MT-ROB-tail MT))))))

(defthm not-INST-at-rob-tail-if-execute-or-complete
    (implies  (and (inv MT MA)
                   (INST-in i MT) (INST-p i)
                   (MAETT-p MT) (MA-state-p MA)
                   (or (execute-stg-p (INST-stg i))
                       (complete-stg-p (INST-stg i)))
                   (not (b1p (ROB-full? (MA-ROB MA)))))
              (equal (equal (INST-tag i) (MT-rob-tail MT)) nil))
  :hints (("Goal" :in-theory (disable robe-valid-MT-ROB-tail-if-not-rob-full)
                  :use (:instance robe-valid-MT-ROB-tail-if-not-rob-full))))

;;; End of Lemmas related to rob-empty?

;;;;; Lemmas about other fields in robe;;;;;;;
(defthm not-INST-bu-if-decode-error
    (implies (and (b1p (INST-decode-error? i))
                  (inst-p i))
             (equal (INST-bu? i) 0))
  :hints (("goal" :in-theory (enable INST-decode-error? INST-bu?
                                     decode-illegal-inst?
                                     equal-b1p-converter
                                     INST-opcode
                                     lift-b-ops logbit* decode rdb
                                     INST-cntlv))))

(defthm INST-BU-if-INST-decode-error-detected
    (implies (and (INST-p i)
                  (INST-decode-error-detected-p i))
             (equal (INST-BU? i) 0))
  :hints (("Goal" :in-theory (enable INST-BU? INST-decode-error-detected-p
                                     INST-cntlv equal-b1p-converter
                                     lift-b-ops decode logbit*
                                     rdb))))

(defthm INST-BU-if-INST-data-accs-error-detected
    (implies (and (INST-p i)
                  (INST-data-accs-error-detected-p i))
             (equal (INST-BU? i) 0))
  :hints (("Goal" :in-theory (enable INST-BU? INST-data-accs-error-detected-p
                                     INST-load-accs-error-detected-p
                                     INST-store-accs-error-detected-p
```

```
                                    INST-cntlv equal-b1p-converter
                                    lift-b-ops decode logbit*
                                    rdb))))

(encapsulate nil
(local
(defthm robe-branch-INST-branch-help
    (implies (and (MA-state-p MA)
                  (MAETT-p MT)
                  (rob-index-p rix)
                  (INST-inv (inst-of-tag rix MT) MA)
                  (uniq-inst-of-tag rix MT)
                  (not (INST-fetch-error-detected-p (inst-of-tag rix MT)))
                  (not (b1p (INST-modified? (inst-of-tag rix MT))))
                  (not (b1p (inst-specultv? (inst-of-tag rix MT)))))
             (equal (robe-branch? (nth-robe rix (MA-rob MA)))
                    (INST-BU? (inst-of-tag rix MT))))
  :Hints (("goal" :in-theory (enable inst-inv-def)
                  :use (:instance INST-is-at-one-of-the-stages
                                  (i (inst-of-tag rix MT)))))))

(local
(defthm robe-branch-INST-branch-help2
    (implies (and (MA-state-p MA)
                  (MAETT-p MT)
                  (rob-index-p rix)
                  (INST-inv (inst-of-tag rix MT) MA)
                  (uniq-inst-of-tag rix MT)
                  (INST-fetch-error-detected-p (inst-of-tag rix MT))
                  (not (b1p (INST-modified? (inst-of-tag rix MT))))
                  (not (b1p (inst-specultv? (inst-of-tag rix MT)))))
             (equal (robe-branch? (nth-robe rix (MA-rob MA)))
                    0))
  :Hints (("goal" :in-theory (enable inst-inv-def)
                  :use (:instance INST-is-at-one-of-the-stages
                                  (i (inst-of-tag rix MT)))))))

(defthm robe-branch-INST-branch-1
    (implies (and (inv MT MA)
                  (MA-state-p MA)
                  (MAETT-p MT)
                  (rob-index-p rix)
                  (uniq-inst-of-tag rix MT)
                  (not (b1p (INST-modified? (inst-of-tag rix MT))))
                  (not (b1p (inst-specultv? (inst-of-tag rix MT)))))
             (equal (robe-branch? (nth-robe rix (MA-rob MA)))
                    (if (INST-excpt-detected-p (inst-of-tag rix MT))
                        0 (INST-BU? (inst-of-tag rix MT)) )))
  :hints (("Goal" :in-theory (enable exception-relations
                                     INST-EXCPT-DETECTED-P))))

(defthm robe-branch-INST-branch-2
    (implies (and (inv MT MA)
                  (MA-state-p MA)
                  (MAETT-p MT)
                  (INST-p i)
                  (INST-in i MT)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (dispatched-p i)
                  (not (committed-p i)))
             (equal (robe-branch? (nth-robe (INST-tag i) (MA-rob MA)))
```

```
                  (if (INST-excpt-detected-p i)
                      0 (INST-BU? i)))))
)

(encapsulate nil
(local
(defthm robe-val-INST-dest-val-1-help
    (implies (and (MAETT-p MT) (MA-state-p MA)
                  (complete-stg-p (INST-stg (inst-of-tag rix MT)))
                  (rob-index-p rix)
                  (INST-inv (inst-of-tag rix MT) MA)
                  (uniq-inst-of-tag rix MT)
                  (INST-writeback-p (inst-of-tag rix MT))
                  (not (b1p (INST-modified? (inst-of-tag rix MT))))
                  (not (b1p (inst-specultv? (inst-of-tag rix MT))))
                  (not (INST-excpt-detected-p (inst-of-tag rix MT))))
             (equal (robe-val (nth-robe rix (MA-rob MA)))
                    (INST-dest-val (inst-of-tag rix MT))))
  :hints (("goal" :in-theory (enable INST-inv
                                     INST-excpt-detected-p
                                     INST-inv-def)))))

(defthm robe-val-INST-dest-val-1
    (implies (and (inv MT MA)
                  (complete-stg-p (INST-stg (inst-of-tag rix MT)))
                  (INST-writeback-p (inst-of-tag rix MT))
                  (MAETT-p MT) (MA-state-p MA)
                  (rob-index-p rix)
                  (uniq-inst-of-tag rix MT)
                  (not (b1p (INST-modified? (inst-of-tag rix MT))))
                  (not (b1p (inst-specultv? (inst-of-tag rix MT))))
                  (not (INST-excpt-detected-p (inst-of-tag rix MT))))
             (equal (robe-val (nth-robe rix (MA-rob MA)))
                    (INST-dest-val (inst-of-tag rix MT)))))

(defthm robe-val-INST-dest-val-2
    (implies (and (inv MT MA)
                  (complete-stg-p (INST-stg i))
                  (INST-writeback-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (not (INST-excpt-detected-p i)))
             (equal (robe-val (nth-robe (INST-tag i) (MA-rob MA)))
                    (INST-dest-val i))))
)

(defthm robe-dest-INST-dest-reg-if-complete
    (implies (and (inv MT MA)
                  (complete-stg-p (INST-stg i))
                  (INST-writeback-p i)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (not (INST-excpt-detected-P I))
                  (inst-p i) (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (robe-dest (nth-robe (INST-tag i) (MA-ROB MA)))
                    (INST-dest-reg i)))
  :Hints (("goal" :in-theory (e/d (INST-inv
                                   complete-inst-inv
                                   INST-EXCPT-DETECTED-P
```

```
                                    complete-inst-robe-inv)
                                   (INST-inv-if-inst-in))
                  :use ((:instance INST-inv-if-inst-in)))))

(defthm robe-dest-INST-dest-reg-if-complete-2
    (implies (and (inv MT MA)
                  (complete-stg-p (INST-stg (inst-of-tag rix MT)))
                  (INST-writeback-p (inst-of-tag rix MT))
                  (not (b1p (INST-modified? (inst-of-tag rix MT))))
                  (not (b1p (inst-specultv? (inst-of-tag rix MT))))
                  (not (INST-excpt-detected-P (inst-of-tag rix MT)))
                  (uniq-inst-of-tag rix MT)
                  (rob-index-p rix)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (robe-dest (nth-robe rix (MA-ROB MA)))
                    (INST-dest-reg (inst-of-tag rix MT))))
  :Hints (("goal" :in-theory (e/d () (robe-dest-INST-dest-reg-if-complete))
                  :use (:instance robe-dest-INST-dest-reg-if-complete
                                  (i (inst-of-tag rix MT))))))

(defthm robe-val-INST-br-target-1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (rob-index-p rix)
                  (uniq-inst-of-tag rix MT)
                  (b1p (INST-bu? (inst-of-tag rix MT)))
                  (not (b1p (inst-specultv? (inst-of-tag rix MT))))
                  (not (b1p (INST-modified? (inst-of-tag rix MT))))
                  (not (INST-fetch-error-detected-p (inst-of-tag rix MT))))
             (equal (ROBE-val (nth-robe rix (MA-ROB MA)))
                    (INST-br-target (inst-of-tag rix MT))))
  :hints (("goal" :in-theory (e/d (INST-inv
                                   INST-excpt-detected-p
                                   INST-inv-def)
                                  (INST-INV-INST-OF-TAG
                                   INST-INV-IF-INST-IN))
                  :use (:instance INST-INV-INST-OF-TAG))))

(defthm robe-val-INST-br-target-2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (dispatched-p i)
                  (not (committed-p i))
                  (b1p (INST-bu? i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i)))
             (equal (ROBE-val (nth-robe (INST-tag i) (MA-ROB MA)))
                    (INST-br-target i)))
  :hints (("goal" :use (:instance robe-val-INST-br-target-1
                                  (rix (INST-tag i))))))

(defthm INST-sync-if-INST-data-accs-error-detected
    (implies (and (INST-p i)
                  (INST-data-accs-error-detected-p i))
             (equal (INST-sync? i) 0))
  :hints (("Goal" :in-theory (enable INST-sync? INST-data-accs-error-detected-p
                                     INST-load-accs-error-detected-p
                                     INST-store-accs-error-detected-p
                                     INST-cntlv equal-b1p-converter
                                     lift-b-ops decode logbit*
```

```
                                    rdb))))

(encapsulate nil
(local
(defthm robe-sync-INST-sync-1-help
    (implies (and (MAETT-p MT)
                  (INST-inv (inst-of-tag rix MT) MA)
                  (MA-state-p MA)
                  (rob-index-p rix)
                  (uniq-inst-of-tag rix MT)
                  (not (INST-fetch-error-detected-p (inst-of-tag rix MT)))
                  (not (b1p (INST-modified? (inst-of-tag rix MT))))
                  (not (b1p (inst-specultv? (inst-of-tag rix MT)))))
             (equal (robe-sync? (nth-robe rix (MA-rob MA)))
                    (INST-sync? (inst-of-tag rix MT))))
  :hints (("Goal" :In-theory (enable inst-inv-def)))))

(defthm robe-sync-INST-sync-1
    (implies (and (inv MT MA)
                  (MA-state-p MA)
                  (MAETT-p MT)
                  (rob-index-p rix)
                  (uniq-inst-of-tag rix MT)
                  (not (INST-fetch-error-detected-p (inst-of-tag rix MT)))
                  (not (b1p (INST-modified? (inst-of-tag rix MT))))
                  (not (b1p (inst-specultv? (inst-of-tag rix MT)))))
             (equal (robe-sync? (nth-robe rix (MA-rob MA)))
                    (INST-sync? (inst-of-tag rix MT))))
  :hints (("Goal" :in-theory (enable exception-relations))))

(defthm robe-sync-INST-sync-2
    (implies (and (inv MT MA)
                  (MA-state-p MA)
                  (MAETT-p MT)
                  (INST-p i)
                  (INST-in i MT)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (not (INST-fetch-error-detected-p i))
                  (dispatched-p i)
                  (not (committed-p i)))
             (equal (robe-sync? (nth-robe (INST-tag i) (MA-rob MA)))
                    (INST-sync? i)))
  :hints (("Goal" :in-theory (enable exception-relations))))
)

(defthm robe-excpt-INST-excpt-flags-1
    (implies (and (inv MT MA)
                  (b1p (robe-complete? (nth-robe rix (MA-rob MA))))
                  (not (b1p (inst-specultv? (inst-of-tag rix MT))))
                  (not (b1p (INST-modified? (inst-of-tag rix MT))))
                  (MA-state-p MA)
                  (MAETT-p MT)
                  (rob-index-p rix)
                  (uniq-inst-of-tag rix MT))
             (equal (robe-excpt (nth-robe rix (MA-rob MA)))
                    (INST-excpt-flags (inst-of-tag rix MT))))
  :hints (("goal" :use (:instance INST-inv-inst-of-tag)
                  :in-theory (e/d (inst-inv-def)
                                  (INST-inv-inst-of-tag
                                   INST-INV-IF-INST-IN)))))
```

```
(defthm robe-excpt-INST-excpt-flags-2
    (implies (and (inv MT MA)
                  (complete-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MA-state-p MA)
                  (INST-p i)
                  (MAETT-p MT)
                  (INST-in i MT))
             (equal (robe-excpt (nth-robe (INST-tag i) (MA-rob MA)))
                    (INST-excpt-flags i))))

(defthm robe-wb-INST-wb-1
    (implies (and (inv MT MA)
                  (MA-state-p MA)
                  (MAETT-p MT)
                  (rob-index-p rix)
                  (uniq-inst-of-tag rix MT)
                  (not (INST-fetch-error-detected-p (inst-of-tag rix MT)))
                  (not (b1p (INST-modified? (inst-of-tag rix MT))))
                  (not (b1p (inst-specultv? (inst-of-tag rix MT)))))
             (equal (robe-wb? (nth-robe rix (MA-rob MA)))
                    (INST-wb? (inst-of-tag rix MT))))
  :hints (("goal" :use (:instance INST-inv-inst-of-tag)
                  :in-theory (e/d (inst-inv-def)
                                  (INST-inv-inst-of-tag
                                   INST-INV-IF-INST-IN)))))

(defthm robe-wb-INST-wb-2
    (implies (and (inv MT MA)
                  (MA-state-p MA)
                  (INST-p i)
                  (MAETT-p MT)
                  (INST-in i MT)
                  (not (INST-fetch-error-detected-p i))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (dispatched-p i)
                  (not (committed-p i)))
             (equal (robe-wb? (nth-robe (INST-tag i) (MA-rob MA)))
                    (INST-wb? i))))

(defthm robe-wb-sreg-INST-wb-sreg-1
    (implies (and (inv MT MA)
                  (MA-state-p MA)
                  (MAETT-p MT)
                  (rob-index-p rix)
                  (uniq-inst-of-tag rix MT)
                  (not (INST-fetch-error-detected-p (inst-of-tag rix MT)))
                  (not (b1p (INST-modified? (inst-of-tag rix MT))))
                  (not (b1p (inst-specultv? (inst-of-tag rix MT)))))
             (equal (robe-wb-sreg? (nth-robe rix (MA-rob MA)))
                    (INST-wb-sreg? (inst-of-tag rix MT))))
  :hints (("goal" :use (:instance INST-inv-inst-of-tag)
                  :in-theory (e/d (inst-inv-def)
                                  (INST-inv-inst-of-tag
                                   INST-INV-IF-INST-IN)))))

(defthm robe-wb-sreg-INST-cntlv-wb-sreg-2
    (implies (and (inv MT MA)
                  (MA-state-p MA)
                  (MAETT-p MT)
```

```
                (INST-p i)
                (INST-in i MT)
                (not (INST-fetch-error-detected-p i))
                (not (b1p (INST-modified? i)))
                (not (b1p (inst-specultv? i)))
                (dispatched-p i)
                (not (committed-p i)))
           (equal (robe-wb-sreg? (nth-robe (INST-tag i) (MA-rob MA)))
                  (INST-wb-sreg? i))))

(defthm INST-wb-if-INST-data-accs-error-detected
    (implies (and (INST-p i)
                  (INST-data-accs-error-detected-p i))
             (equal (INST-rfeh? i) 0))
  :hints (("Goal" :in-theory (enable INST-rfeh? INST-data-accs-error-detected-p
                                     INST-load-accs-error-detected-p
                                     INST-store-accs-error-detected-p
                                     INST-cntlv equal-b1p-converter
                                     lift-b-ops decode logbit*
                                     rdb))))


(defthm robe-rfeh-INST-rfeh-1
    (implies (and (inv MT MA)
                  (MA-state-p MA)
                  (MAETT-p MT)
                  (rob-index-p rix)
                  (uniq-inst-of-tag rix MT)
                  (not (INST-fetch-error-detected-p (inst-of-tag rix MT)))
                  (not (b1p (INST-modified? (inst-of-tag rix MT))))
                  (not (b1p (inst-specultv? (inst-of-tag rix MT)))))
             (equal (robe-rfeh? (nth-robe rix (MA-rob MA)))
                    (INST-rfeh? (inst-of-tag rix MT))))
  :hints (("goal" :use ((:instance INST-inv-inst-of-tag)
                        (:instance INST-is-at-one-of-the-stages
                                   (i (inst-of-tag rix MT))))
                  :in-theory (e/d (INST-inv-opener
                                   exception-relations
                                   INST-excpt-detected-p)
                                  (INST-inv-inst-of-tag
                                   INST-INV-IF-INST-IN)))))

(defthm robe-rfeh-INST-rfeh-2
    (implies (and (inv MT MA)
                  (MA-state-p MA)
                  (MAETT-p MT)
                  (INST-p i)
                  (INST-in i MT)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (not (INST-fetch-error-detected-p i))
                  (dispatched-p i)
                  (not (committed-p i)))
             (equal (robe-rfeh? (nth-robe (INST-tag i) (MA-rob MA)))
                    (INST-rfeh? i))))

(defthm robe-br-predict-INST-br-predict-1
    (implies (and (inv MT MA)
                  (MA-state-p MA)
                  (MAETT-p MT)
                  (rob-index-p rix)
                  (uniq-inst-of-tag rix MT)
```

```
                (not (INST-fetch-error-detected-p (inst-of-tag rix MT)))
                (not (b1p (INST-modified? (inst-of-tag rix MT))))
                (not (b1p (inst-specultv? (inst-of-tag rix MT)))))
           (equal (robe-br-predict? (nth-robe rix (MA-rob MA)))
                  (INST-br-predict? (inst-of-tag rix MT))))
  :hints (("goal" :use (:instance INST-inv-inst-of-tag)
                  :in-theory (e/d (inst-inv-def)
                                  (INST-inv-inst-of-tag
                                   INST-INV-IF-INST-IN)))))

(defthm robe-br-predict-INST-br-predict-2
    (implies (and (inv MT MA)
                  (MA-state-p MA)
                  (MAETT-p MT)
                  (INST-p i)
                  (INST-in i MT)
                  (not (INST-fetch-error-detected-p i))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (dispatched-p i)
                  (not (committed-p i)))
             (equal (robe-br-predict? (nth-robe (INST-tag i) (MA-rob MA)))
                    (INST-br-predict? i))))

; The value in the INST-br-actual field
(defthm robe-br-predict-INST-br-actual-1
    (implies (and (inv MT MA)
                  (MA-state-p MA)
                  (MAETT-p MT)
                  (rob-index-p rix)
                  (uniq-inst-of-tag rix MT)
                  (b1p (inst-BU? (inst-of-tag rix MT)))
                  (complete-stg-p (INST-stg (inst-of-tag rix MT)))
                  (not (INST-fetch-error-detected-p (inst-of-tag rix MT)))
                  (not (b1p (INST-modified? (inst-of-tag rix MT))))
                  (not (b1p (inst-specultv? (inst-of-tag rix MT)))))
             (equal (robe-br-actual? (nth-robe rix (MA-rob MA)))
                    (INST-branch-taken? (inst-of-tag rix MT))))
  :hints (("goal" :use (:instance INST-inv-inst-of-tag)
                  :in-theory (e/d (inst-inv-def)
                                  (INST-inv-inst-of-tag
                                   INST-INV-IF-INST-IN)))))

(defthm robe-br-predict-INST-br-actual-2
    (implies (and (inv MT MA)
                  (MA-state-p MA)
                  (MAETT-p MT)
                  (INST-p i)
                  (INST-in i MT)
                  (b1p (inst-BU? i))
                  (complete-stg-p (INST-stg i))
                  (not (INST-fetch-error-detected-p i))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (or (execute-stg-p (INST-stg i))
                      (complete-stg-p (INST-stg i))))
             (equal (robe-br-actual? (nth-robe (INST-tag i) (MA-rob MA)))
                    (INST-branch-taken? i))))

(defthm INST-branch-INST-sync-exclusive
    (implies (and (inv MT MA)
                  (MA-state-p MA)
```

```
                (MAETT-p MT)
                (rob-index-p rix)
                (b1p (robe-valid? (nth-robe rix (MA-rob MA)))))
           (not (and (b1p (INST-BU? (inst-of-tag rix MT)))
                     (b1p (INST-sync? (inst-of-tag rix MT))))))
  :hints (("Goal" :in-theory (enable INST-BU? INST-cntlv INST-sync?)))
  :rule-classes
  ((:rewrite :corollary
    (implies (and (inv MT MA)
                (MA-state-p MA)
                (MAETT-p MT)
                (rob-index-p rix)
                (b1p (robe-valid? (nth-robe rix (MA-rob MA))))
                (b1p (INST-BU? (inst-of-tag rix MT))))
           (equal (INST-sync? (inst-of-tag rix MT)) 0))
    :hints (("goal" :in-theory (enable equal-b1p-converter lift-b-ops))))))

(defthm INST-sync-INST-BU-exclusive-2
    (implies (and (INST-p i) (b1p (INST-BU? i)))
             (equal (INST-sync? i) 0))
  :hints (("goal" :in-theory (enable INST-sync? INST-BU? INST-cntlv))))

(encapsulate nil
(local
(defthm robe-branch-robe-sync-exclusive-help
    (implies (and (consistent-robe-p (nth-robe rix (MA-rob MA))
                                     rix (MA-ROB MA))
                (b1p (robe-valid? (nth-robe rix (MA-ROB MA))))
                (b1p (robe-branch? (nth-robe rix (MA-ROB MA))))
                (MA-state-p MA)
                (rob-index-p rix))
           (equal (robe-sync? (nth-robe rix (MA-ROB MA))) 0))
  :hints (("goal" :in-theory (enable consistent-robe-p
                                     lift-b-ops equal-b1p-converter)))))

; Robe-branch? and robe-sync? field of an ROB entry are not 1 simultaneously.
(defthm robe-branch-robe-sync-exclusive
    (implies (and (inv MT MA)
                (b1p (robe-valid? (nth-robe rix (MA-ROB MA))))
                (b1p (robe-branch? (nth-robe rix (MA-ROB MA))))
                (rob-index-p rix)
                (MAETT-p MT) (MA-state-p MA))
           (equal (robe-sync? (nth-robe rix (MA-ROB MA))) 0))
  :hints (("goal" :use (:instance robe-branch-robe-sync-exclusive-help))))
)

(encapsulate nil
(local
(defthm robe-wb-robe-sync-exclusive-help
    (implies (and (consistent-robe-p (nth-robe rix (MA-rob MA))
                                     rix (MA-ROB MA))
                (b1p (robe-valid? (nth-robe rix (MA-ROB MA))))
                (b1p (robe-wb? (nth-robe rix (MA-ROB MA))))
                (MA-state-p MA)
                (rob-index-p rix))
           (equal (robe-sync? (nth-robe rix (MA-ROB MA))) 0))
  :hints (("goal" :in-theory (enable consistent-robe-p
                                     lift-b-ops equal-b1p-converter)))))

; Robe-wb? and robe-sync? field of an ROB entry are not 1 simultaneously.
(defthm robe-wb-robe-sync-exclusive
    (implies (and (inv MT MA)
```

```
                (b1p (robe-valid? (nth-robe rix (MA-ROB MA))))
                (b1p (robe-wb? (nth-robe rix (MA-ROB MA))))
                (rob-index-p rix)
                (MAETT-p MT) (MA-state-p MA))
           (equal (robe-sync? (nth-robe rix (MA-ROB MA))) 0))
  :hints (("goal" :use (:instance robe-wb-robe-sync-exclusive-help)
                  :in-theory (e/d (equal-b1p-converter lift-b-ops)
                                  (robe-wb-robe-sync-exclusive-help)))))
)

; Robe-pc field of an ROB entry to which instruction i is assigned contains
; the pc value for i.
(defthm robe-pc-INST-tag
    (implies (and (inv MT MA)
                  (dispatched-p i)
                  (not (committed-p i))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (inst-p i) (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (robe-pc (nth-robe (INST-tag i) (MA-ROB MA)))
                    (INST-pc i)))
  :Hints (("goal" :in-theory (e/d (INST-inv inst-inv-def
                                   committed-p dispatched-p
                                                 INST-pc)
                                  (INST-inv-if-inst-in))
                  :use ((:instance INST-inv-if-inst-in)
                        (:instance INST-is-at-one-of-the-stages)))))

(defthm robe-pc-INST-tag-2
    (implies (and (inv MT MA)
                  (uniq-inst-of-tag rix MT)
                  (rob-index-p rix)
                  (not (b1p (INST-modified? (inst-of-tag rix MT))))
                  (not (b1p (inst-specultv? (inst-of-tag rix MT))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (robe-pc (nth-robe rix (MA-ROB MA)))
                    (INST-pc (inst-of-tag rix MT))))
  :Hints (("goal" :use (:instance robe-pc-INST-tag
                                  (i (inst-of-tag rix MT))))))

; Field robe-dest of an ROB entry contains the index to the destination
; register of the corresponding instruction.
(defthm robe-dest-INST-dest-reg
    (implies (and (inv MT MA)
                  (INST-writeback-p i)
                  (dispatched-p i)
                  (not (committed-p i))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (not (INST-fetch-error-detected-P I))
                  (inst-p i) (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (robe-dest (nth-robe (INST-tag i) (MA-ROB MA)))
                    (INST-dest-reg i)))
  :Hints (("goal" :in-theory (e/d (INST-inv
                                   complete-inst-inv
                                   complete-inst-robe-inv
                                   execute-inst-inv
                                   execute-inst-robe-inv
                                   committed-p dispatched-p)
                                  (INST-inv-if-inst-in))
```

```
                :use ((:instance INST-inv-if-inst-in)
                      (:instance INST-is-at-one-of-the-stages)))))

(defthm robe-dest-INST-dest-reg-2
    (implies (and (inv MT MA)
                  (INST-writeback-p (inst-of-tag rix MT))
                  (not (b1p (INST-modified? (inst-of-tag rix MT))))
                  (not (b1p (inst-specultv? (inst-of-tag rix MT))))
                  (not (INST-fetch-error-detected-P (inst-of-tag rix MT)))
                  (uniq-inst-of-tag rix MT)
                  (rob-index-p rix)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (robe-dest (nth-robe rix (MA-ROB MA)))
                    (INST-dest-reg (inst-of-tag rix MT))))
  :hints (("goal" :in-theory (disable robe-dest-INST-dest-reg)
                  :use (:instance robe-dest-INST-dest-reg
                                  (i (inst-of-tag rix MT))))))

(defthm robe-branch-nth-robe-if-inst-fetch-error
    (implies (and (inv MT MA)
                  (dispatched-p i)
                  (not (committed-p i))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (INST-FETCH-ERROR-DETECTED-P I)
                  (inst-p i) (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (robe-branch? (nth-robe (INST-tag i) (MA-ROB MA))) 0))
  :Hints (("goal" :in-theory (e/d (INST-inv
                                   complete-inst-inv
                                   complete-inst-robe-inv
                                   execute-inst-inv
                                   execute-inst-robe-inv
                                   committed-p dispatched-p
                                   INST-dest-reg)
                                  (INST-inv-if-inst-in))
                  :use ((:instance INST-inv-if-inst-in)
                        (:instance INST-is-at-one-of-the-stages)))))

(defthm robe-sync-nth-robe-if-inst-fetch-error
    (implies (and (inv MT MA)
                  (dispatched-p i)
                  (not (committed-p i))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (INST-FETCH-ERROR-DETECTED-P I)
                  (inst-p i) (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (robe-sync? (nth-robe (INST-tag i) (MA-ROB MA))) 0))
  :Hints (("goal" :in-theory (e/d (INST-inv
                                   complete-inst-inv
                                   complete-inst-robe-inv
                                   execute-inst-inv
                                   execute-inst-robe-inv
                                   committed-p dispatched-p
                                   INST-dest-reg)
                                  (INST-inv-if-inst-in))
                  :use ((:instance INST-inv-if-inst-in)
                        (:instance INST-is-at-one-of-the-stages)))))

(defthm robe-rfeh-nth-robe-if-inst-fetch-error
    (implies (and (inv MT MA)
```

```
                (dispatched-p i)
                (not (committed-p i))
                (not (b1p (INST-modified? i)))
                (not (b1p (inst-specultv? i)))
                (INST-FETCH-ERROR-DETECTED-P I)
                (inst-p i) (INST-in i MT)
                (MAETT-p MT) (MA-state-p MA))
           (equal (robe-rfeh? (nth-robe (INST-tag i) (MA-ROB MA))) 0))
  :Hints (("goal" :in-theory (e/d (INST-inv
                                   complete-inst-inv
                                   complete-inst-robe-inv
                                   execute-inst-inv
                                   execute-inst-robe-inv
                                   committed-p dispatched-p
                                   INST-dest-reg)
                                  (INST-inv-if-inst-in))
                  :use ((:instance INST-inv-if-inst-in)
                        (:instance INST-is-at-one-of-the-stages)))))

(defthm robe-rfeh-nth-robe-if-inst-fetch-error-2
    (implies (and (inv MT MA)
                  (uniq-inst-of-tag rix MT)
                  (INST-FETCH-ERROR-DETECTED-P (inst-of-tag rix MT))
                  (not (b1p (INST-modified? (inst-of-tag rix MT))))
                  (not (b1p (inst-specultv? (inst-of-tag rix MT))))
                  (rob-index-p rix)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (robe-rfeh? (nth-robe rix (MA-ROB MA))) 0))
  :Hints (("goal" :use (:instance robe-rfeh-nth-robe-if-inst-fetch-error
                                  (i (inst-of-tag rix MT))))))

(defthm robe-excpt-nth-robe-if-fetch-or-decode-error
    (implies (and (inv MT MA)
                  (dispatched-p i)
                  (not (committed-p i))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (or (b1p (inst-fetch-error? i))
                      (b1p (inst-decode-error? i)))
                  (inst-p i) (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (robe-excpt (nth-robe (INST-tag i) (MA-ROB MA)))
                    (INST-excpt-flags i)))
  :Hints (("goal" :in-theory (e/d (INST-inv
                                   complete-inst-inv
                                   complete-inst-robe-inv
                                   execute-inst-inv
                                   execute-inst-robe-inv
                                   committed-p dispatched-p
                                   INST-dest-reg)
                                  (INST-inv-if-inst-in))
                  :use ((:instance INST-inv-if-inst-in)
                        (:instance INST-is-at-one-of-the-stages)))))

;;; Lemmas to relate inst-of-tag and INST-at-stg
(defthm uniq-inst-of-tag-if-IU-RS0-valid
    (implies (and (inv MT MA)
                  (b1p (RS-valid? (IU-RS0 (MA-IU MA))))
                  (MA-state-p MA) (MAETT-p MT))
             (uniq-inst-of-tag (RS-tag (IU-RS0 (MA-IU MA))) MT))
  :hints (("goal" :in-theory (enable IU-RS-field-INST-at-lemmas))))
```

```
(defthm inst-of-tag-IU-RS0-tag-INST-at-stg-IU-RS0
    (implies (and (inv MT MA)
                  (b1p (RS-valid? (IU-RS0 (MA-IU MA))))
                  (MA-state-p MA) (MAETT-p MT))
             (equal (inst-of-tag (RS-tag (IU-RS0 (MA-IU MA))) MT)
                    (INST-at-stg '(IU RS0) MT)))
  :hints (("goal" :in-theory (enable IU-RS-field-INST-at-lemmas))))

(defthm uniq-inst-of-tag-if-IU-RS1-valid
    (implies (and (inv MT MA)
                  (b1p (RS-valid? (IU-RS1 (MA-IU MA))))
                  (MA-state-p MA) (MAETT-p MT))
             (uniq-inst-of-tag (RS-tag (IU-RS1 (MA-IU MA))) MT))
  :hints (("goal" :in-theory (enable IU-RS-field-INST-at-lemmas))))


(defthm inst-of-tag-IU-RS-tag-INST-at-stg-IU-RS1
    (implies (and (inv MT MA)
                  (b1p (RS-valid? (IU-RS1 (MA-IU MA))))
                  (MA-state-p MA) (MAETT-p MT))
             (equal (inst-of-tag (RS-tag (IU-RS1 (MA-IU MA))) MT)
                    (INST-at-stg '(IU RS1) MT)))
  :hints (("goal" :in-theory (enable IU-RS-field-INST-at-lemmas))))

(defthm uniq-inst-of-tag-if-BU-RS0-valid
    (implies (and (inv MT MA)
                  (b1p (BU-RS-valid? (BU-RS0 (MA-BU MA))))
                  (MA-state-p MA) (MAETT-p MT))
             (uniq-inst-of-tag (BU-RS-tag (BU-RS0 (MA-BU MA))) MT))
  :hints (("goal" :in-theory (enable BU-RS-field-inst-at-lemmas))))

(defthm uniq-inst-of-tag-if-BU-RS1-valid
    (implies (and (inv MT MA)
                  (b1p (BU-RS-valid? (BU-RS1 (MA-BU MA))))
                  (MA-state-p MA) (MAETT-p MT))
             (uniq-inst-of-tag (BU-RS-tag (BU-RS1 (MA-BU MA))) MT))
  :hints (("goal" :in-theory (enable BU-RS-field-inst-at-lemmas))))

;;; definition and lemmas about tag-in-order
(defun tag-in-order (rix1 rix2 MT)
  (if (b1p (MT-rob-flg MT))
      (or (and (<= (MT-rob-head MT) rix1)
               (< rix2 (MT-rob-tail MT)))
          (and (<= (MT-rob-head MT) rix1) (< rix1 rix2))
          (and (< rix1 rix2) (< rix2 (MT-rob-tail MT))))
      (< rix1 rix2)))
(in-theory (disable tag-in-order))

;;;;;  End of Lemmas about ROB and INST ;;;;;

;;;;;  Instruction Specific properties about INST fields
; Relations between INST-wrong-branch? and INST-BU?
(defthm not-INST-wrong-branch-if-not-INST-BU
    (implies (and (INST-p I) (not (b1p (INST-BU? i))))
             (not (b1p (INST-wrong-branch? i))))
  :hints (("goal" :in-theory (enable INST-wrong-branch?
                                     bv-eqv-iff-equal
                                     INST-opcode
                                     INST-cntlv decode
                                     rdb logbit*
                                     INST-BU? lift-b-ops))))
```

```
(defthm INST-sync-if-INST-rfeh
    (implies (and (INST-p i) (b1p (INST-rfeh? i)))
             (equal (INST-sync? i) 1))
  :hints (("goal" :in-theory (enable INST-sync? INST-rfeh? decode
                                     INST-cntlv logbit* rdb))))

(defthm INST-sync-opcode-5-8
    (implies (INST-p i)
             (iff (b1p (INST-sync? i))
                  (or (equal (INST-opcode i) 5)
                      (equal (INST-opcode i) 8))))
  :hints (("goal" :in-theory (enable INST-sync? RDB logbit* decode INST-opcode
                                     INST-cntlv lift-b-ops)))
  :rule-classes nil)

;; INST-sync? field for a load store instruction.
(defthm not-INST-sync-if-INST-LSU?
    (implies (and (INST-p i) (b1p (INST-LSU? i)))
             (equal (INST-sync? i) 0))
  :hints (("Goal" :in-theory (enable INST-LSU? INST-cntlv decode
                                     lift-b-ops equal-b1p-converter
                                     INST-sync?
                                     logbit* rdb))))

;; INST-rfeh field for a load store instruction.
(defthm not-INST-rfeh-if-INST-LSU?
    (implies (and (INST-p i) (b1p (INST-LSU? i)))
             (equal (INST-rfeh? i) 0))
  :hints (("Goal" :in-theory (enable INST-LSU? INST-cntlv decode
                                     lift-b-ops equal-b1p-converter
                                     INST-rfeh? logbit* rdb))))

;; INST-wb field for a store instruction
(defthm not-INST-wb-if-INST-store?
    (implies (and (INST-p i) (b1p (INST-store? i)))
             (equal (INST-wb? i) 0))
  :hints (("Goal" :in-theory (enable INST-store? INST-cntlv decode
                                     lift-b-ops equal-b1p-converter
                                     INST-LSU? INST-ld-st?
                                     INST-wb? logbit* rdb))))

;; INST-wb-sreg field for a load store instruction
(defthm not-INST-wb-sreg-if-INST-LSU?
    (implies (and (INST-p i) (b1p (INST-LSU? i)))
             (equal (INST-wb-sreg? i) 0))
  :hints (("Goal" :in-theory (enable INST-LSU? INST-cntlv decode
                                     lift-b-ops equal-b1p-converter
                                     INST-wb-sreg? logbit* rdb))))

(deflabel begin-opcode-inst-type)
(defthm INST-IU-opcode-0
    (implies (and (INST-p i) (equal (INST-opcode i) 0))
             (equal (INST-IU? i) 1))
  :hints (("goal" :in-theory (enable INST-IU? INST-cntlv decode
                                     logbit* rdb))))

(defthm INST-MU-opcode-1
    (implies (and (INST-p i) (equal (INST-opcode i) 1))
             (equal (INST-MU? i) 1))
  :hints (("goal" :in-theory (enable INST-MU? INST-cntlv decode
                                     logbit* rdb))))
```

```
(defthm INST-BU-opcode-2
    (implies (and (INST-p i) (equal (INST-opcode i) 2))
             (equal (INST-BU? i) 1))
  :hints (("goal" :in-theory (enable INST-BU? INST-cntlv decode
                                     logbit* rdb))))

(defthm INST-load-opcode-3
    (implies (and (INST-p i) (equal (INST-opcode i) 3))
             (equal (INST-load? i) 1))
  :hints (("goal" :in-theory (enable INST-load? INST-cntlv decode
                                     INST-LSU? INST-LD-ST?
                                     logbit* rdb))))
(defthm INST-store-opcode-4
    (implies (and (INST-p i) (equal (INST-opcode i) 4))
             (equal (INST-store? i) 1))
  :hints (("goal" :in-theory (enable INST-store? INST-cntlv decode
                                     INST-LSU? INST-LD-ST?
                                     logbit* rdb))))

(defthm INST-load-opcode-6
    (implies (and (INST-p i) (equal (INST-opcode i) 6))
             (equal (INST-load? i) 1))
  :hints (("goal" :in-theory (enable INST-load? INST-cntlv decode
                                     INST-LSU? INST-LD-ST?
                                     logbit* rdb))))

(defthm INST-store-opcode-7
    (implies (and (INST-p i) (equal (INST-opcode i) 7))
             (equal (INST-store? i) 1))
  :hints (("goal" :in-theory (enable INST-store? INST-cntlv decode
                                     INST-LSU? INST-LD-ST?
                                     logbit* rdb))))

(deflabel end-opcode-inst-type)
(deftheory opcode-inst-type
    (set-difference-theories
     (universal-theory 'end-opcode-inst-type)
     (universal-theory 'begin-opcode-inst-type)))

(in-theory (disable opcode-inst-type))

(defthm opcodes-at-IU-stg-p
    (implies (and (inv MT MA)
                  (IU-stg-p (INST-stg i))
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (or (equal (INST-opcode i) 0)
                 (equal (INST-opcode i) 9)
                 (equal (INST-opcode i) 10)))
  :hints (("goal" :in-theory (e/d (equal-b1p-converter
                                   decode rdb logbit* lift-b-ops
                                   INST-IU? inst-cntlv)
                                  (INST-IU-IF-IU-STG-P ))
                  :use (:instance INST-IU-IF-IU-STG-P)))
  :rule-classes nil)

(defthm opcode-2-at-BU-stg-p
    (implies (and (inv MT MA)
                  (BU-stg-p (INST-stg i))
                  (INST-in i MT) (INST-p i)
```

```
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i))))
           (equal (INST-opcode i) 2))
  :hints (("goal" :in-theory (e/d (equal-b1p-converter
                                   decode rdb logbit* lift-b-ops
                                   INST-BU? inst-cntlv)
                                  (INST-BU-IF-BU-STG-P ))
                  :use (:instance INST-BU-IF-BU-STG-P)))
  :rule-classes nil)

(defthm opcode-1-at-MU-stg-p
    (implies (and (inv MT MA)
                  (MU-stg-p (INST-stg i))
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (equal (INST-opcode i) 1))
  :hints (("goal" :in-theory (e/d (equal-b1p-converter
                                   decode rdb logbit* lift-b-ops
                                   INST-MU? inst-cntlv)
                                  (INST-MU-IF-MU-STG-P ))
                  :use (:instance INST-MU-IF-MU-STG-P)))
  :rule-classes nil)

(defthm opcodes-at-LSU-stg-p
    (implies (and (inv MT MA)
                  (LSU-stg-p (INST-stg i))
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (or (equal (INST-opcode i) 3)
                 (equal (INST-opcode i) 4)
                 (equal (INST-opcode i) 6)
                 (equal (INST-opcode i) 7)))
  :hints (("goal" :in-theory (e/d (equal-b1p-converter
                                   decode rdb logbit* lift-b-ops
                                   INST-LSU? inst-cntlv)
                                  (INST-LSU-IF-LSU-STG-P ))
                  :use (:instance INST-LSU-IF-LSU-STG-P)))
  :rule-classes nil)

(defthm INST-type-exclusive
    (and (implies (and (INST-p i) (b1p (INST-LSU? i)))
                  (equal (INST-no-unit? i) 0))
         (implies (and (INST-p i) (b1p (INST-BU? i)))
                  (equal (INST-no-unit? i) 0))
         (implies (and (INST-p i) (b1p (INST-MU? i)))
                  (equal (INST-no-unit? i) 0))
         (implies (and (INST-p i) (b1p (INST-IU? i)))
                  (equal (INST-no-unit? i) 0))
         (implies (and (INST-p i) (b1p (INST-LSU? i)))
                  (equal (INST-IU? i) 0))
         (implies (and (INST-p i) (b1p (INST-BU? i)))
                  (equal (INST-IU? i) 0))
         (implies (and (INST-p i) (b1p (INST-MU? i)))
                  (equal (INST-IU? i) 0))
         (implies (and (INST-p i) (b1p (INST-LSU? i)))
                  (equal (INST-MU? i) 0))
         (implies (and (INST-p i) (b1p (INST-BU? i)))
```

```
                (equal (INST-MU? i) 0))
         (implies (and (INST-p i) (b1p (INST-LSU? i)))
                (equal (INST-BU? i) 0)))
  :hints (("goal" :in-theory (enable INST-LSU? INST-no-unit?
                                     INST-MU? INST-IU? INST-BU?
                                     INST-cntlv
                                     equal-b1p-converter lift-b-ops))))

(defthm INST-type-exclusive-2
    (and (implies (and (INST-p i) (b1p (INST-LSU? i)))
                  (not (b1p (INST-no-unit? i))))
         (implies (and (INST-p i) (b1p (INST-BU? i)))
                  (not (b1p (INST-no-unit? i))))
         (implies (and (INST-p i) (b1p (INST-MU? i)))
                  (not (b1p (INST-no-unit? i))))
         (implies (and (INST-p i) (b1p (INST-IU? i)))
                  (not (b1p (INST-no-unit? i))))
         (implies (and (INST-p i) (b1p (INST-no-unit? i)))
                  (not (b1p (INST-IU? i))))
         (implies (and (INST-p i) (b1p (INST-LSU? i)))
                  (not (b1p (INST-IU? i))))
         (implies (and (INST-p i) (b1p (INST-BU? i)))
                  (not (b1p (INST-IU? i))))
         (implies (and (INST-p i) (b1p (INST-MU? i)))
                  (not (b1p (INST-IU? i))))
         (implies (and (INST-p i) (b1p (INST-no-unit? i)))
                  (not (b1p (INST-MU? i))))
         (implies (and (INST-p i) (b1p (INST-IU? i)))
                  (not (b1p (INST-MU? i))))
         (implies (and (INST-p i) (b1p (INST-LSU? i)))
                  (not (b1p (INST-MU? i))))
         (implies (and (INST-p i) (b1p (INST-BU? i)))
                  (not (b1p (INST-MU? i))))
         (implies (and (INST-p i) (b1p (INST-no-unit? i)))
                  (not (b1p (INST-BU? i))))
         (implies (and (INST-p i) (b1p (INST-IU? i)))
                  (not (b1p (INST-BU? i))))
         (implies (and (INST-p i) (b1p (INST-MU? i)))
                  (not (b1p (INST-BU? i))))
         (implies (and (INST-p i) (b1p (INST-LSU? i)))
                  (not (b1p (INST-BU? i))))
         (implies (and (INST-p i) (b1p (INST-no-unit? i)))
                  (not (b1p (INST-LSU? i))))
         (implies (and (INST-p i) (b1p (INST-IU? i)))
                  (not (b1p (INST-LSU? i))))
         (implies (and (INST-p i) (b1p (INST-MU? i)))
                  (not (b1p (INST-LSU? i))))
         (implies (and (INST-p i) (b1p (INST-BU? i)))
                  (not (b1p (INST-LSU? i))))))

(in-theory (disable INST-type-exclusive))

(defthm INST-store-INST-sync-exclusive
    (implies (and (INST-p i) (b1p (INST-sync? i)))
             (equal (INST-store? i) 0))
  :hints (("goal" :in-theory (enable INST-sync? INST-store?
                                     INST-LSU? lift-b-ops equal-b1p-converter
                                     INST-cntlv decode logbit* rdb))))

(defthm INST-store-INST-rfeh-exclusive
    (implies (and (INST-p i) (b1p (INST-rfeh? i)))
             (equal (INST-store? i) 0))
```

```
  :hints (("goal" :in-theory (enable INST-rfeh? INST-store?
                                     INST-LSU? lift-b-ops equal-b1p-converter
                                     INST-cntlv decode logbit* rdb))))

(defthm INST-load-INST-store-exclusive
    (and (implies (b1p (INST-store? i)) (not (b1p (INST-load? i))))
         (implies (b1p (INST-load? i)) (not (b1p (INST-store? i)))))
  :hints (("goal" :in-theory (enable INST-load? INST-store? lift-b-ops))))

(encapsulate nil
(local
(defthm not-INST-sync-if-at-complete-wbuf-help
    (implies (and (INST-p i)
                  (MA-state-p MA)
                  (INST-inv i MA)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (wbuf-stg-p (INST-stg i)))
             (equal (INST-sync? i) 0))
  :hints (("goal" :in-theory (enable inst-inv-def wbuf-stg-p)))))

; INST-sync? for the instruction at write buffer.
(defthm not-INST-sync-if-at-complete-wbuf
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (MAETT-p MT)
                  (INST-p i)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (MA-state-p MA)
                  (wbuf-stg-p (INST-stg i)))
             (equal (INST-sync? i) 0)))
) ;encapsulate

(encapsulate nil
(local
(defthm not-INST-rfeh-if-at-complete-wbuf-help
    (implies (and (INST-p i)
                  (MA-state-p MA)
                  (INST-inv i MA)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (wbuf-stg-p (INST-stg i)))
             (equal (INST-rfeh? i) 0))
  :hints (("Goal" :in-theory (enable wbuf-stg-p inst-inv-def)))))

;  INST-rfeh field for an instruction in the write buffer.
(defthm not-INST-rfeh-if-at-complete-wbuf
    (implies (and (inv MT MA)
                  (INST-p i)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (INST-in i MT)
                  (wbuf-stg-p (INST-stg i)))
             (equal (INST-rfeh? i) 0)))
)  ;encapsulate

(encapsulate nil
(local
(defthm not-INST-wb-if-at-complete-wbuf-help
```

```
    (implies (and (INST-p i)
                  (MA-state-p MA)
                  (INST-inv i MA)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (wbuf-stg-p (INST-stg i)))
             (equal (INST-wb? i) 0))
  :hints (("goal" :in-theory (enable wbuf-stg-p inst-inv-def)))))

; INST-wb field for an instruction in the write buffer.
(defthm not-INST-wb-if-at-complete-wbuf
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (INST-p i)
                  (wbuf-stg-p (INST-stg i)))
             (equal (INST-wb? i) 0)))
)

(encapsulate nil
(local
(defthm not-INST-wb-sreg-if-at-complete-wbuf-help
    (implies (and (INST-p i)
                  (MA-state-p MA)
                  (INST-inv i MA)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (wbuf-stg-p (INST-stg i)))
             (equal (INST-wb-sreg? i) 0))
  :hints (("Goal" :in-theory (enable inst-inv-def wbuf-stg-p)))))

; INST-wb-sreg for an instruction in a write buffer.
(defthm not-INST-wb-sreg-if-at-complete-wbuf
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (INST-p i)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (wbuf-stg-p (INST-stg i)))
             (equal (INST-wb-sreg? i) 0)))
)

(encapsulate nil
(local
(defthm not-INST-branch-if-at-complete-wbuf-help
    (implies (and (INST-p i)
                  (MA-state-p MA)
                  (INST-inv i MA)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (wbuf-stg-p (INST-stg i)))
             (equal (INST-BU? i) 0))
  :hints (("Goal" :in-theory (enable wbuf-stg-p inst-inv-def
                                     equal-b1p-converter)))))

;  INST-branch field for an instruction in a write buffer.
(defthm not-INST-branch-if-at-complete-wbuf
```

```
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (MA-state-p MA)
                  (MAETT-p MT)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (INST-p i)
                  (wbuf-stg-p (INST-stg i)))
             (equal (INST-BU? i) 0)))
)

(defthm INST-writeback-p-INST-BU-exclusive
    (implies (and (INST-p i) (b1p (INST-BU? I)))
             (not (INST-writeback-p i)))
  :hints (("goal" :in-theory (enable INST-writeback-p INST-BU?
                                     inst-cntlv INST-opcode))))

;;;;; Relations about INST predicates such as INST-cause-jmp, INST-exintr-now
;;;;; INST-start-specultv?
(defthm not-INST-cause-jmp-if-not-flush-all
    (implies (not (b1p (flush-all? MA sigs)))
             (equal (INST-cause-jmp? i MT MA sigs) 0))
  :hints (("goal" :in-theory (enable INST-cause-jmp? flush-all?
                                     equal-b1p-converter
                                     lift-b-ops))))

(defthm not-INST-exintr-now-if-not-flush-all
    (implies (not (b1p (flush-all? MA sigs)))
             (equal (INST-exintr-now? i MA sigs) 0))
  :hints (("goal" :in-theory (enable INST-exintr-now? flush-all?
                                     equal-b1p-converter ex-intr?
                                     lift-b-ops))))

(defthm not-INST-exintr-now-if-not-fetch-inst
    (implies (b1p (fetch-inst? MA sigs))
             (equal (INST-exintr-now? i MA sigs) 0))
  :hints (("goal" :in-theory (enable lift-b-ops fetch-inst?))))

(defthm not-INST-cause-jmp-if-not-fetch-inst
    (implies (b1p (fetch-inst? MA sigs))
             (equal (INST-cause-jmp? i MT MA sigs) 0))
  :hints (("goal" :in-theory (enable lift-b-ops fetch-inst?))))

(defthm not-execute-stg-p-if-ex-intr
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i)
                  (INST-in i MT)
                  (b1p (ex-intr? MA sigs)))
             (not (execute-stg-p (INST-stg i))))
  :hints (("Goal" :in-theory (enable ex-intr? lift-b-ops))))

(defthm not-complete-stg-p-if-ex-intr
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i)
                  (INST-in i MT)
                  (b1p (ex-intr? MA sigs)))
             (not (complete-stg-p (INST-stg i))))
    :hints (("Goal" :in-theory (enable ex-intr? lift-b-ops))))

(defthm inst-exintr-now-if-ex-intr
```

```
    (implies (and (b1p (ex-intr? MA sigs))
                  (or (IFU-stg-p (INST-stg I))
                      (DQ-stg-p (INST-stg I))))
             (equal (INST-exintr-now? i MA sigs) 1))
  :hints (("Goal" :in-theory (enable inst-exintr-now? equal-b1p-converter
                                     ex-intr?
                                     lift-b-ops)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (and (b1p (ex-intr? MA sigs))
                           (not (dispatched-p i))
                           (INST-p i))
                      (equal (INST-exintr-now? i MA sigs) 1))
             :hints (("goal" :in-theory (enable dispatched-p))))))

(defthm INST-exintr-now-if-ex-intr-if-not-commit
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (committed-p i))
                  (b1p (ex-intr? MA sigs))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-exintr-now? i MA sigs) 1))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (disable inst-is-at-one-of-the-stages))))

(defthm not-INST-exintr-now-if-commit-inst?
    (implies (and (inv MT MA)
                  (MA-state-p MA)
                  (MAETT-p MT)
                  (b1p (commit-inst? MA)))
             (equal (INST-exintr-now? i MA sigs) 0))
  :hints (("Goal" :in-theory (enable commit-inst? INST-exintr-now?
                                     lift-b-ops equal-b1p-converter))))

; INST-exintr-now? is true for an instruction only if
; an external interrupt signal is set or the external interrupt flag
; has been set.
(defthm INST-exintr-now-when-no-exintr-flag
    (implies (and (MA-state-p MA)
                  (MA-input-p sigs)
                  (not (b1p (MA-input-exintr sigs)))
                  (not (b1p (exintr-flag? MA))))
             (not (b1p (INST-exintr-now? i MA sigs))))
  :hints (("Goal" :in-theory (enable INST-exintr-now? lift-b-ops
                                     exintr-flag?))))

; The external interrupt is not asserted if there is an instruction in
; the execution or complete stage.
(defthm not-ex-intr-if-INST-at-execute-or-complete-stg
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (INST-p i)
                  (dispatched-p i)
                  (not (committed-p i)))
             (equal (ex-intr? MA sigs) 0))
  :hints (("goal" :in-theory (e/d (ex-intr? lift-b-ops equal-b1p-converter)
                                  (not-ex-intr-if-rob-not-empty))
                  :use (:instance not-ex-intr-if-rob-not-empty))))
```

```
(in-theory (disable not-ex-intr-if-INST-at-execute-or-complete-stg))

;; INST-excpt-flags is the current exception flags stored in the ROB.
;; This is equivalent to INST-excpt-detected-p
(defthm excpt-raised-INST-excpt
    (implies (INST-p i)
             (equal (excpt-raised? (INST-excpt-flags i))
                    (if (INST-excpt-detected-p i) 1 0)))
  :hints (("Goal" :in-theory (enable INST-excpt-flags INST-excpt-detected-p))))

; Only instructions at IFU or DQ stage can be externally interrupted.
(defthm not-INST-exintr-now-if-not-IFU-or-dq
    (implies (dispatched-p i)
             (equal (INST-exintr-now? i MA sigs) 0))
  :hints (("Goal" :in-theory (enable INST-exintr-now?))))

; INST-exintr-now? predicate is not true if the external interrupt
; is not detected by ex-intr?.
(defthm not-INST-exintr-now-if-not-ex-intr
    (implies (and (inv MT MA)
                  (not (b1p (ex-intr? MA sigs))))
             (equal (INST-exintr-now? i MA sigs) 0))
  :hints (("Goal" :in-theory (enable INST-exintr-now? lift-b-ops
                                     equal-b1p-converter
                                     ex-intr?))))

; INST-start-specultv cannot be true for committed instructions.
; INST-start-specultv is true if it is followed by instructions that
; will eventually thrown away.
(defthm not-INST-start-specultv-if-committed
    (implies (committed-p i)
             (equal (INST-start-specultv? i) 0))
  :hints (("Goal" :in-theory (enable INST-start-specultv? committed-p))))

(defthm INST-start-specultv-if-INST-excpt
    (implies (and (not (committed-p i))
                  (b1p (INST-excpt? i)))
             (equal (INST-start-specultv? i) 1))
  :hints (("goal" :in-theory (enable INST-start-specultv? committed-p
                                     lift-b-ops equal-b1p-converter))))

(defthm INST-start-specultv-if-INST-context-sync
    (implies (and (not (committed-p i))
                  (b1p (INST-context-sync? i)))
             (equal (INST-start-specultv? i) 1))
  :hints (("goal" :in-theory (enable INST-start-specultv? committed-p
                                     lift-b-ops equal-b1p-converter))))

(defthm INST-start-specultv-if-INST-wrong-branch?
    (implies (and (not (committed-p i))
                  (b1p (INST-wrong-branch? i)))
             (equal (INST-start-specultv? i) 1))
  :hints (("goal" :in-theory (enable INST-start-specultv? committed-p
                                     lift-b-ops equal-b1p-converter))))

(defthm not-INST-cause-jmp-if-robe-not-head
    (implies (and (INST-p i)
                  (MAETT-p MT)
                  (not (equal (INST-tag i) (MT-rob-head MT))))
             (equal (INST-cause-jmp? i MT MA sigs) 0))
  :hints (("Goal" :in-theory (enable INST-cause-jmp? lift-b-ops
                                     equal-b1p-converter))))
```

```
;; INST-cause-jmp can be true only if the instruction is at complete stage.
(defthm not-INST-cause-jmp-if-not-complete
    (implies (not (complete-stg-p (INST-stg i)))
             (equal (INST-cause-jmp? i MT MA sigs) 0))
  :hints (("Goal" :in-theory (enable INST-cause-jmp?))))

;; INST-cause-jmp? is not true if commit-inst? is not asserted.
(defthm not-INST-cause-jmp-if-not-commit-inst
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (INST-p i)
                  (INST-in i MT)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (commit-inst? MA))))
             (equal (INST-cause-jmp? i MT MA sigs) 0))
  :hints (("Goal" :in-theory (e/d (INST-cause-jmp? lift-b-ops
                                   equal-b1p-converter
                                   exception-relations
                                   commit-jmp? commit-inst?
                                   leave-excpt? enter-excpt?)
                                  ())
                  :cases ((INST-excpt-detected-p i)))))

; INST-cause-jmp? and ex-intr? are exclusive.
(defthm not-INST-cause-jmp-if-ex-intr
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (INST-p i)
                  (b1p (ex-intr? MA sigs)))
             (equal (INST-cause-jmp? i MT MA sigs) 0))
  :hints (("Goal" :use
                  (:instance not-ex-intr-if-INST-at-execute-or-complete-stg)
                  :in-theory (enable INST-cause-jmp? lift-b-ops
                                     equal-b1p-converter))))

(defthm not-commit-jmp-if-rob-head-complete-wbuf
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (INST-p i)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (equal (INST-tag i) (MT-rob-head MT))
                  (or (equal (INST-stg i) '(complete wbuf0))
                      (equal (INST-stg i) '(complete wbuf1))))
             (not (b1p (commit-jmp? MA))))
  :hints (("Goal" :in-theory (enable commit-jmp? lift-b-ops wbuf-stg-p)))
  :rule-classes
  ((:rewrite :corollary
             (implies (and (inv MT MA)
                           (INST-in i MT)
                           (MAETT-p MT)
                           (MA-state-p MA)
                           (INST-p i)
                           (b1p (commit-jmp? MA))
                           (not (b1p (inst-specultv? i)))
```

```
                          (not (b1p (INST-modified? i)))
                          (or (equal (INST-stg i) '(complete wbuf0))
                              (equal (INST-stg i) '(complete wbuf1))))
                     (equal (equal (INST-tag i) (MT-rob-head MT)) nil)))))

(defthm not-leave-excpt-if-rob-head-complete-wbuf
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (INST-p i)
                  (equal (INST-tag i) (MT-rob-head MT))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (or (equal (INST-stg i) '(complete wbuf0))
                      (equal (INST-stg i) '(complete wbuf1))))
             (not (b1p (leave-excpt? MA))))
  :hints (("Goal" :in-theory (enable leave-excpt? lift-b-ops wbuf-stg-p)))
  :rule-classes
  ((:rewrite :corollary
             (implies (and (inv MT MA)
                           (INST-in i MT)
                           (MAETT-p MT)
                           (MA-state-p MA)
                           (INST-p i)
                           (not (b1p (inst-specultv? i)))
                           (not (b1p (INST-modified? i)))
                           (b1p (leave-excpt? MA))
                           (or (equal (INST-stg i) '(complete wbuf0))
                               (equal (INST-stg i) '(complete wbuf1))))
                      (equal (equal (INST-tag i) (MT-rob-head MT)) nil)))))

; If flush-all? is not on, an instruction remains in a MAETT after
; MAETT-update.
(encapsulate nil
(local
(defthm INST-in-step-INST-if-not-flush-all-help
    (implies (and (not (b1p (flush-all? MA sigs)))
                  (member-equal i trace))
             (member-equal (step-INST i MT MA sigs)
                           (step-trace trace MT MA sigs
                                       ISA smc spc)))
  :hints (("goal" :in-theory (enable flush-all? INST-exintr-now? ex-intr?
                                     INST-cause-jmp?
                                     lift-b-ops)))))

(defthm INST-in-step-INST-if-not-flush-all
    (implies (and (not (b1p (flush-all? MA sigs)))
                  (INST-in i MT))
             (INST-in (step-INST i MT MA sigs) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable INST-in MT-step))))
)

;;;;;;;;;;;;;Other lemmas about MT and MA;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defthm LSU-pending-writes-if-wbuf-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (or (equal (INST-stg i) '(commit wbuf0))
                      (equal (INST-stg i) '(commit wbuf1)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-pending-writes? (MA-LSU MA)) 1))
```

```
  :hints (("goal" :in-theory (enable LSU-pending-writes?
                                     equal-b1p-converter wbuf-stg-p
                                     lift-b-ops))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Property about reachable microarchitectural states
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defthm not-ex-intr-if-commit-inst?
    (implies (and (inv MT MA)
                  (MA-state-p MA)
                  (MAETT-p MT)
                  (b1p (commit-inst? MA)))
             (equal (ex-intr? MA sigs) 0))
  :hints (("Goal" :in-theory (enable commit-inst? ex-intr? lift-b-ops
                                     equal-b1p-converter))))

(defthm commit-inst-if-commit-jmp
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (commit-jmp? MA)))
             (equal (commit-inst? MA) 1))
  :hints (("goal" :in-theory (enable commit-jmp? commit-inst? lift-b-ops
                                     equal-b1p-converter))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  step-INST and stage again
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defthm not-commit-stg-step-INST-if-robe-not-head
    (implies (and (INST-p i)
                  (MA-state-p MA)
                  (MAETT-p MT)
                  (inv MT MA)
                  (not (equal (INST-tag i) (MT-rob-head MT)))
                  (not (commit-stg-p (INST-stg i)))
                  (not (retire-stg-p (INST-stg i))))
             (not (commit-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("goal" :in-theory (enable MT-def lift-b-ops))))

(defthm not-retire-stg-step-INST-if-robe-not-head
    (implies (and (INST-p i)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (inv MT MA)
                  (not (equal (INST-tag i) (MT-rob-head MT)))
                  (not (commit-stg-p (INST-stg i)))
                  (not (retire-stg-p (INST-stg i))))
             (not (retire-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("goal" :in-theory (enable MT-def lift-b-ops))))

;;
(defthm not-commit-stg-p-step-INST-if-not-commit-inst?
    (implies (and (not (b1p (commit-inst? MA)))
                  (not (commit-stg-p (INST-stg i))))
             (not (commit-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("goal" :in-theory (enable MT-def lift-b-ops))))

(defthm not-retire-stg-p-step-INST-if-not-commit-inst?
    (implies (and (not (b1p (commit-inst? MA)))
                  (not (commit-stg-p (INST-stg i)))
                  (not (retire-stg-p (INST-stg i))))
             (not (retire-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("goal" :in-theory (enable MT-def lift-b-ops))))
```

```
(defthm retire-INST-stg-step-INST-if-complete-robe-is-head
    (implies (and (INST-p i)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (inv MT MA)
                  (equal (INST-tag i) (MT-rob-head MT))
                  (complete-stg-p (INST-stg i))
                  (b1p (commit-inst? MA))
                  (not (commit-stg-p (INST-stg (step-INST i MT MA sigs)))))
             (retire-stg-p (INST-stg (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (enable MT-def lift-b-ops))))

;; If j is at the complete stage and it does not commit in this cycle,
;; it stays in the complete stage in the next cycle.
(defthm complete-stg-p-step-INST-if-not-INST-commit
    (implies (and (complete-stg-p (INST-stg j))
                  (not (b1p (INST-commit? j MA))))
             (complete-stg-p (INST-stg (step-INST j MT MA sigs))))
  :hints (("Goal" :in-theory (enable step-INST step-INST-complete
                                     lift-b-ops))))

(defthm commit-if-inst-commit
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (b1p (INST-commit? i MA))
                  (not (commit-stg-p (INST-stg (step-INST i MT MA sigs))))
                  (MAETT-p MT) (MA-state-p MA))
             (retire-stg-p (INST-stg (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (enable INST-commit? lift-b-ops)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (INST-in i MT) (INST-p i)
                           (b1p (INST-commit? i MA))
                           (MAETT-p MT) (MA-state-p MA))
                      (committed-p (step-INST i MT MA sigs)))
             :hints (("goal" :in-theory (enable committed-p))))))

(defthm not-commit-if-not-INST-commit
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (not (committed-p i))
                  (not (b1p (INST-commit? i MA))))
             (not (committed-p (step-INST i MT MA sigs))))
  :hints (("Goal" :in-theory (enable step-inst-complete-inst lift-b-ops
                                     step-inst-low-level-functions
                                     committed-p*))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; no-commit-inst-p no-dispatched-inst-p
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defthm no-commit-inst-p-if-no-dispatched-inst-p
    (implies (and (INST-listp trace) (no-dispatched-inst-p trace))
             (no-commit-inst-p trace))
  :hints (("goal" :in-theory (enable dispatched-p))))

(defthm no-trace-CMI-p-if-no-commit-inst-p
    (implies (and (INST-listp trace) (no-commit-inst-p trace))
             (not (trace-CMI-p trace))))
```

```
(encapsulate nil
(local
  (defthm no-dispatched-inst-p-cdr-help
      (implies (and (in-order-trace-p trace)
                    (INST-listp trace)
                    (INST-listp sub)
                    (tail-p sub trace)
                    (not (dispatched-p (car sub))))
               (no-dispatched-inst-p (cdr sub)))
    :hints (("goal" :in-theory (enable dispatched-p)))))

(defthm no-dispatched-inst-p-cdr
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-listp trace)
                  (subtrace-p trace MT)
                  (not (dispatched-p (car trace))))
             (no-dispatched-inst-p (cdr trace)))
  :hints (("Goal" :in-theory (enable inv weak-inv
                                     in-order-dispatch-commit-p
                                     subtrace-p))))
)

(encapsulate nil
(local
 (defthm no-commit-inst-p-cdr-help
     (implies (and (in-order-trace-p trace)
                   (INST-listp trace)
                   (INST-listp sub)
                   (tail-p sub trace)
                   (not (committed-p (car sub))))
              (no-commit-inst-p (cdr sub)))
   :hints (("Goal" :in-theory (enable committed-p)))))


(defthm no-commit-inst-p-cdr
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-listp trace)
                  (subtrace-p trace MT)
                  (not (committed-p (car trace))))
             (no-commit-inst-p (cdr trace)))
  :hints (("goal" :in-theory (enable inv weak-inv
                                     in-order-dispatch-commit-p
                                     subtrace-p))))
)


(defun trace-1st-non-commit-inst (trace)
  (if (endp trace) nil
      (if (not (committed-p (car trace)))
          (car trace)
          (trace-1st-non-commit-inst (cdr trace)))))

(defun MT-1st-non-commit-INST (MT)
  (trace-1st-non-commit-inst (MT-trace MT)))

(defun trace-all-committed-p (trace)
  (if (endp trace) T
      (if (committed-p (car trace))
          (trace-all-committed-p (cdr trace))
          nil)))
```

```
(defun MT-all-committed-p (MT)  (trace-all-committed-p (MT-trace MT)))
(in-theory (disable MT-all-committed-p MT-1st-non-commit-inst))

(encapsulate nil
(local
(defthm not-MT-all-committed-p-if-non-commit-INST-in-help
    (implies (and (member-equal i trace)
                  (not (committed-p i)))
             (not (trace-all-committed-p trace)))))

(defthm not-MT-all-committed-p-if-non-commit-INST-in
    (implies (and (INST-in i MT) (INST-p i)
                  (not (committed-p i)))
             (not (MT-all-committed-p MT)))
  :hints (("goal" :in-theory (enable INST-in MT-all-committed-p))))
)

(encapsulate nil
(local
(defthm INST-p-MT-1st-non-commit-inst-help
    (implies (and (INST-listp trace)
                  (not (trace-all-committed-p trace)))
             (INST-p (trace-1st-non-commit-inst trace)))))

(defthm INST-p-MT-1st-non-commit-inst
    (implies (and (MAETT-p MT)
                  (not (MT-all-committed-p MT)))
             (INST-p (MT-1st-non-commit-inst MT)))
  :hints (("goal" :in-theory
                  (enable MT-all-committed-p MT-1st-non-commit-inst))))
)

(encapsulate nil
(local
(defthm INST-in-MT-1st-non-commit-inst-help
    (implies (not (trace-all-committed-p trace))
             (member-equal (trace-1st-non-commit-inst trace) trace))))

(defthm INST-in-MT-1st-non-commit-inst
    (implies (and (inv MT MA)
                  (not (MT-all-committed-p MT)))
             (INST-in (MT-1st-non-commit-inst MT) MT))
  :hints (("goal" :in-theory (enable MT-all-committed-p MT-1st-non-commit-inst
                                     INST-in))))
)

(encapsulate nil
(local
(defthm committed-p-MT-1st-non-commit-inst-help
    (implies (not (trace-all-committed-p trace))
             (not (committed-p (trace-1st-non-commit-inst trace))))))

(defthm committed-p-MT-1st-non-commit-inst
    (implies (not (MT-all-committed-p MT))
             (not (committed-p (MT-1st-non-commit-inst MT))))
  :hints (("goal" :in-theory
                  (enable MT-1st-non-commit-inst MT-all-committed-p))))
)

(encapsulate nil
```

```
(local
 (defthm inst-specultv-MT-1st-non-commit-inst-help
     (implies (and (INST-listp trace)
                   (trace-correct-speculation-p trace)
                   (not (trace-all-committed-p trace)))
              (equal (inst-specultv? (trace-1st-non-commit-inst trace)) 0))
   :hints (("goal" :in-theory (enable equal-b1p-converter)))))

(defthm inst-specultv-MT-1st-non-commit-inst
    (implies (and (inv MT MA)
                  (not (MT-all-committed-p MT))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (inst-specultv? (MT-1st-non-commit-inst MT)) 0))
  :hints (("goal" :in-theory (enable inv correct-speculation-p
                                     MT-all-committed-p
                                     MT-1st-non-commit-inst))))
)

(encapsulate nil
(local
(defthm INST-tag-MT-1st-non-commit-inst-help
    (implies (and (in-order-rob-trace-p trace rix)
                  (INST-listp trace)
                  (dispatched-p (trace-1st-non-commit-inst trace))
                  (not (trace-all-committed-p trace)))
             (equal (INST-tag (trace-1st-non-commit-inst trace))
                    rix))
  :hints (("goal" :in-theory (enable committed-p)))))

(defthm INST-tag-MT-1st-non-commit-inst
    (implies (and (inv MT MA)
                  (not (MT-all-committed-p MT))
                  (dispatched-p (MT-1st-non-commit-inst MT))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-tag (MT-1st-non-commit-inst MT))
                    (MT-ROB-head MT)))
  :hints (("goal" :in-theory (enable MT-1st-non-commit-inst MT-all-committed-p
                                     inv IN-order-ROB-p))))
)

(encapsulate nil
(local
(defthm dispatched-p-MT-1st-non-commit-inst-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (member-equal i trace) (INST-p i)
                  (INST-listp trace)
                  (not (committed-p i))
                  (dispatched-p i)
                  (MAETT-p MT) (MA-state-p MA))
             (dispatched-p (trace-1st-non-commit-inst trace)))
  :hints (("goal" :in-theory (disable INST-IN-ORDER-DISPATCHED-UNDISPATCHED))
          (when-found (dispatched-p (car trace))
                      (:use (:instance INST-IN-ORDER-DISPATCHED-UNDISPATCHED
                                       (j (car trace)))))))))

(defthm dispatched-p-MT-1st-non-commit-inst
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (committed-p i))
                  (dispatched-p i)
                  (MAETT-p MT) (MA-state-p MA))
```

```
            (dispatched-p (MT-1st-non-commit-inst MT)))
  :hints (("goal" :in-theory (enable MT-1st-non-commit-inst
                                     INST-in))))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; A theory about the instruction at ROB head
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(encapsulate nil
(local
(defthm uniq-inst-of-tag-if-commit-inst-help
    (implies (and (inv MT MA)
                  (no-tag-conflict-at (MT-ROB-head MT) MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (commit-inst? MA)))
             (uniq-inst-of-tag (MT-ROB-head MT) MT))
  :hints (("goal" :in-theory (enable no-tag-conflict-at
                                     commit-inst? lift-b-ops)))
  :rule-classes nil))

;  If an instruction commits this cycle, there is a unique instruction
; stored at the head of ROB.
(defthm uniq-inst-of-tag-if-commit-inst
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (commit-inst? MA)))
             (uniq-inst-of-tag (MT-ROB-head MT) MT))
  :hints (("goal" :use (:instance uniq-inst-of-tag-if-commit-inst-help))))
)

(defthm uniq-inst-of-tag-if-context-sync
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (or (b1p (commit-jmp? MA))
                      (b1p (leave-excpt? MA))
                      (b1p (enter-excpt? MA))))
             (uniq-inst-of-tag (MT-rob-head MT) MT))
  :hints (("goal" :in-theory (enable commit-jmp? lift-b-ops
                                     enter-excpt? leave-excpt?))))

(defthm complete-inst-of-tag-if-context-sync
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (or (b1p (commit-jmp? MA))
                      (b1p (leave-excpt? MA))
                      (b1p (enter-excpt? MA))))
             (complete-stg-p (INST-stg (inst-of-tag (MT-rob-head MT) MT))))
  :hints (("goal" :in-theory (e/d (commit-jmp? lift-b-ops
                                               leave-excpt? enter-excpt?
                                               committed-p dispatched-p)
                                  (inst-of-tag-is-dispatched
                                   inst-of-tag-is-not-committed))
                  :use ((:instance inst-of-tag-is-not-committed
                                   (rix (MT-rob-head MT)))
                        (:instance inst-of-tag-is-dispatched
                                   (rix (MT-rob-head MT)))))))

(theory-invariant
 (not (and (or (member-equal '(:rewrite uniq-inst-of-tag-if-context-sync)
                             theory)
               (member-equal '(:rewrite complete-inst-of-tag-if-context-sync)
                             theory))
```

```
          (or (member-equal '(:definition enter-excpt?) theory)
              (member-equal '(:definition commit-jmp?) theory)
              (member-equal '(:definition leave-excpt?) theory)))))

; The instruction at the head of ROB will not be in execute-stg if
; commit-inst? is on.
(defthm not-execute-stg-p-inst-at-rob-head-if-commit-inst
    (implies (and (inv MT MA)
                  (MA-state-p MA) (MAETT-p MT)
                  (b1p (commit-inst? MA)))
     (not (execute-stg-p (INST-stg (inst-of-tag (MT-ROB-head MT) MT)))))
  :hints (("goal" :in-theory (enable commit-inst? lift-b-ops))))

(defthm complete-inst-of-tag-if-commit-inst
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (commit-inst? MA)))
             (complete-stg-p (INST-stg (inst-of-tag (MT-rob-head MT) MT))))
  :hints (("goal" :in-theory (e/d (commit-inst? lift-b-ops
                                                committed-p dispatched-p)
                                  (inst-of-tag-is-dispatched
                                   inst-of-tag-is-not-committed))
                  :use ((:instance inst-of-tag-is-not-committed
                                   (rix (MT-rob-head MT)))
                        (:instance inst-of-tag-is-dispatched
                                   (rix (MT-rob-head MT)))))))

(theory-invariant
 (not (and (member-equal '(:definition commit-inst?) theory)
           (or (member-equal '(:rewrite uniq-inst-of-tag-if-commit-inst)
                             theory)
               (member-equal '(:rewrite COMPLETE-INST-OF-TAG-IF-COMMIT-INST)
                             theory)
               (member-equal '(:rewrite
                         NOT-EXECUTE-STG-P-INST-AT-ROB-HEAD-IF-COMMIT-INST)
                         theory)))))

(deftheory incompatible-with-excpt-in-MAETT-lemmas
    '(uniq-inst-of-tag-if-commit-inst
      uniq-inst-of-tag-if-context-sync
      complete-inst-of-tag-if-context-sync
      COMPLETE-INST-OF-TAG-IF-COMMIT-INST
      NOT-EXECUTE-STG-P-INST-AT-ROB-HEAD-IF-COMMIT-INST))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Theory about MT-non-commit-trace and MT-non-retire-trace.
;;  MT-non-commit-trace and MT-non-retire-trace return the subtrace
;;  of a MAETT which contains non-committed or non-retired instructions.

;; Definitions of MT-non-commit-trace and MT-non-retire-trace.
(defun non-commit-trace (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace) nil
      (if (or (commit-stg-p (INST-stg (car trace)))
              (retire-stg-p (INST-stg (car trace))))
          (non-commit-trace (cdr trace))
          trace)))

; MT-non-commit-trace returns the largest subtrace of an MAETT
; which begins with an non-committed instruction.  Since instructions
; commit in program order, a subtrace returned by MT-non-commit-trace
; contains only non-committed instructions.
```

```
(defun MT-non-commit-trace (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (non-commit-trace (MT-trace MT)))
(in-theory (disable MT-non-commit-trace))

(defun non-retire-trace (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace) nil
      (if (retire-stg-p (INST-stg (car trace)))
          (non-retire-trace (cdr trace))
          trace)))

; MT-non-retire-trace returns the largest subtrace of an MAETT which
; begins with an non-retired instructions.  A subtrace returned by
; MT-non-retire-trace may contains non-retired instructions.
(defun MT-non-retire-trace (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (non-retire-trace (MT-trace MT)))
(in-theory (disable MT-non-retire-trace))

;; Basic lemmas about MT-non-commit-trace and MT-non-retire-trace.
(defthm INST-listp-non-commit-trace
    (implies (INST-listp trace)
             (INST-listp (non-commit-trace trace))))

(defthm INST-listp-MT-non-commit-trace
    (implies (MAETT-p MT)
             (INST-listp (MT-non-commit-trace MT)))
  :hints (("Goal" :in-theory (enable MT-non-commit-trace))))

(encapsulate nil
(local
 (defthm subtrace-p-MT-non-commit-trace-help
     (implies (INST-listp trace) (tail-p (non-commit-trace trace) trace))))

(defthm subtrace-p-MT-non-commit-trace
    (implies (MAETT-p MT) (subtrace-p (MT-non-commit-trace MT) MT))
  :hints (("Goal" :in-theory (enable subtrace-p MT-non-commit-trace))))
) ;encapsulate

(encapsulate nil
(local
(defthm not-retire-stg-p-car-MT-non-commit-trace-help
    (implies (and (consp (non-commit-trace trace)) (INST-listp trace))
             (not (retire-stg-p (INST-stg (car (non-commit-trace trace))))))))

; The first entry of an subtrace returned by MT-non-commit-trace
; represents an non-committed instruction.  Thus the represented
; instruction is not in retire-stg-p.
(defthm not-retire-stg-p-car-MT-non-commit-trace
    (implies (and (MAETT-p MT) (consp (MT-non-commit-trace MT)))
             (not (retire-stg-p (INST-stg (car (MT-non-commit-trace MT))))))
  :hints (("Goal" :in-theory (enable MT-non-commit-trace))))
)

(encapsulate nil
(local
(defthm not-retire-stg-p-car-MT-non-retire-trace-help
    (implies (and (consp (non-retire-trace trace)) (INST-listp trace))
             (not (retire-stg-p (INST-stg (car (non-retire-trace trace))))))))

; The first entry of an subtrace returned by MT-non-retire-trace
```

```
; represents a non-retired instruction.
(defthm not-retire-stg-p-car-MT-non-retire-trace
    (implies (and (MAETT-p MT) (consp (MT-non-retire-trace MT)))
             (not (retire-stg-p (INST-stg (car (MT-non-retire-trace MT))))))
  :hints (("Goal" :in-theory (enable MT-non-retire-trace))))
)

(encapsulate nil
(local
(defthm not-commit-stg-p-car-MT-non-commit-trace-help
    (implies (and (consp (non-commit-trace trace)) (INST-listp trace))
             (not (commit-stg-p (INST-stg (car (non-commit-trace trace))))))))

; The first instruction of an subtrace returned by MT-non-commit-trace
; is not committed.
(defthm not-commit-stg-p-car-MT-non-commit-trace
    (implies (and (MAETT-p MT) (consp (MT-non-commit-trace MT)))
             (not (commit-stg-p (INST-stg (car (MT-non-commit-trace MT))))))
  :hints (("Goal" :in-theory (enable MT-non-commit-trace))))
)

; MT-no-commit-trace does not contain any committed instruction entries.
; A similar lemma about MT-no-retire-trace is not true since
; instruction retirement is not in-order.
(encapsulate nil
(local
(defthm no-commit-inst-p-MT-non-commit-trace
    (implies (and (in-order-trace-p trace)
                  (INST-listp trace))
             (no-commit-inst-p (non-commit-trace trace))))
)

(defthm no-commit-trace-p-MT-non-commit-trace
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA))
             (no-commit-inst-p (MT-non-commit-trace MT)))
  :hints (("Goal" :in-theory (enable weak-inv inv
                                     MT-non-commit-trace
                                     IN-ORDER-DISPATCH-COMMIT-P))))
)

(encapsulate nil
(local
 (defthm member-equal-non-commit-inst-MT-non-commit-trace-help
     (implies (and (in-order-trace-p trace)
                   (INST-p i)
                   (member-equal i trace)
                   (not (retire-stg-p (inst-stg i)))
                   (not (commit-stg-p (inst-stg i))))
              (member-equal i (non-commit-trace trace)))))

; A non-committed instruction is a member of MT-non-commit-trace.
(defthm member-equal-non-commit-inst-MT-non-commit-trace
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (not (retire-stg-p (inst-stg i)))
                  (not (commit-stg-p (inst-stg i))))
             (member-equal i (MT-non-commit-trace MT)))
  :hints (("goal" :in-theory (enable MT-non-commit-trace INST-in))))
) ;encapsulate
```

```
(encapsulate nil
(local
 (defthm member-equal-non-retire-inst-MT-non-retire-trace-help
     (implies (and (in-order-trace-p trace)
                   (INST-p i)
                   (member-equal i trace)
                   (not (retire-stg-p (inst-stg i))))
              (member-equal i (non-retire-trace trace)))))

; A non-retired instruction is a member of MT-non-commit-trace.
(defthm member-equal-non-retire-inst-MT-non-retire-trace
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (not (retire-stg-p (inst-stg i))))
             (member-equal i (MT-non-retire-trace MT)))
  :hints (("goal" :in-theory (enable MT-non-retire-trace INST-in))))
)

; trace-correct-speculation-p is true for MT-non-commit-trace.
(encapsulate nil
(local
 (defthm trace-correct-speculation-p-MT-non-commit-trace-help
     (implies (trace-correct-speculation-p trace)
              (trace-correct-speculation-p (non-commit-trace trace)))))

(defthm trace-correct-speculation-p-MT-non-commit-trace
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA))
             (trace-correct-speculation-p (MT-non-commit-trace MT)))
  :hints (("Goal" :in-theory (enable MT-non-commit-trace
                                     inv
                                     correct-speculation-p))))
)

; (MT-non-commit-trace MT) is a subtrace of MT.
(encapsulate nil
(local
(defthm tail-p-MT-non-commit-trace-help
    (implies (INST-listp trace) (tail-p (non-commit-trace trace) trace))))

(defthm tail-p-MT-non-commit-trace
    (implies (MAETT-p MT) (tail-p (MT-non-commit-trace MT) (MT-trace MT)))
  :hints (("goal" :in-theory (enable MT-non-commit-trace))))
)

; (MT-non-retire-trace MT) is a subtrace of MT.
(encapsulate nil
(local
(defthm tail-p-MT-non-retire-trace-help
    (implies (INST-listp trace) (tail-p (non-retire-trace trace) trace))))

(defthm tail-p-MT-non-retire-trace
    (implies (MAETT-p MT) (tail-p (MT-non-retire-trace MT) (MT-trace MT)))
  :hints (("goal" :in-theory (enable MT-non-retire-trace))))
)

(encapsulate nil
(local
 (defthm super-trace-of-MT-non-commit-trace-w-car-commit-help
     (implies (and (tail-p trace trace2)
                   (not (retire-stg-p (INST-stg (car trace))))
                   (not (commit-stg-p (INST-stg (car trace))))
```

```
                (tail-p (non-commit-trace trace2) trace))
           (equal (equal (non-commit-trace trace2) trace) t))
   :hints ((when-found-multiple
            ((TAIL-P TRACE (CDR TRACE2)) (TAIL-P TRACE2 TRACE))
            (:use (:instance tail-p-transitivity
                             (lst1 trace2) (lst2 trace)
                             (lst3 (cdr trace2)))))))

; A subtrace trace is a terminating sublist of MT-non-commit-trace,
; if the first instruction in trace is committed.
(defthm super-trace-of-MT-non-commit-trace-w-car-commit
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (subtrace-p trace MT)
                  (not (retire-stg-p (INST-stg (car trace))))
                  (not (commit-stg-p (INST-stg (car trace))))
                  (tail-p (MT-non-commit-trace MT) trace))
             (equal (MT-non-commit-trace MT) trace))
  :hints (("Goal" :in-theory (enable MT-non-commit-trace
                                     subtrace-p)))
  :rule-classes
  ((:rewrite :corollary
             (implies (and (inv MT MA)
                           (MAETT-p MT)
                           (MA-state-p MA)
                           (subtrace-p trace MT)
                           (not (retire-stg-p (INST-stg (car trace))))
                           (not (commit-stg-p (INST-stg (car trace))))
                           (tail-p (MT-non-commit-trace MT) trace))
                      (equal (equal (MT-non-commit-trace MT) trace)
                             T)))))

) ;encapsulate

(encapsulate nil
(local
 (defthm super-trace-of-MT-non-retire-trace-w-car-retire-help
     (implies (and (tail-p trace trace2)
                   (not (retire-stg-p (INST-stg (car trace))))
                   (tail-p (non-retire-trace trace2) trace))
              (equal (equal (non-retire-trace trace2) trace) t))
   :hints ((when-found-multiple
            ((TAIL-P TRACE (CDR TRACE2)) (TAIL-P TRACE2 TRACE))
            (:use (:instance tail-p-transitivity
                             (lst1 trace2) (lst2 trace)
                             (lst3 (cdr trace2)))))))

; A subtrace trace is a terminating sublist of MT-non-commit-trace,
; if the first instruction in trace is retired.
(defthm super-trace-of-MT-non-retire-trace-w-car-retire
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (subtrace-p trace MT)
                  (not (retire-stg-p (INST-stg (car trace))))
                  (tail-p (MT-non-retire-trace MT) trace))
             (equal (MT-non-retire-trace MT) trace))
  :hints (("Goal" :in-theory (enable MT-non-retire-trace
                                     subtrace-p)))
  :rule-classes
  ((:rewrite :corollary
```

```
             (implies (and (inv MT MA)
                           (MAETT-p MT)
                           (MA-state-p MA)
                           (subtrace-p trace MT)
                           (not (retire-stg-p (INST-stg (car trace))))
                           (tail-p (MT-non-retire-trace MT) trace))
                      (equal (equal (MT-non-retire-trace MT) trace)
                             T)))))
)

(local
 (defthm no-uniq-inst-of-tag-in-trace-if-no-dispatched-inst-p
     (implies (no-dispatched-inst-p trace)
              (not (uniq-inst-of-tag-in-trace idx trace)))))

(encapsulate nil

(local
(defthm not-uniq-inst-of-tag-in-trace-if-car-dispatched-help
    (implies (and (in-order-trace-p trace)
                  (INST-listp sub)
                  (tail-p sub trace)
                  (not (dispatched-p (car sub))))
             (not (uniq-inst-of-tag-in-trace idx (cdr sub))))
  :hints (("goal" :in-theory (disable INST-is-at-one-of-the-stages))
          (when-found (IFU-STG-P (INST-STG (CAR SUB)))
          (:use (:instance INST-is-at-one-of-the-stages (i (car sub))))))))

; If (car trace) is not a dispatched instruction, following instructions
; in (cdr trace) are not dispatched either.  Therefore, no instruction
; in (cdr trace) is stored in the ROB.
(defthm not-uniq-inst-of-tag-in-trace-if-car-dispatched
    (implies (and (inv MT MA)
                  (not (dispatched-p (car trace)))
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace))
             (not (uniq-inst-of-tag-in-trace idx (cdr trace))))
  :hints (("goal" :in-theory (enable inv
                                     subtrace-p
                                     in-order-dispatch-commit-p))))
)

;;; Definition of local predicate MT-all-commit-before. The predicate
;;; returns non-nil value if all the instructions before instruction i are
;;; committed.
(defun trace-all-commit-before (i trace)
  (if (endp trace)
      T
      (if (equal i (car trace))
          T
          (and (or (commit-stg-p (INST-stg (car trace)))
                   (retire-stg-p (INST-stg (car trace))))
               (trace-all-commit-before i (cdr trace))))))

(defun MT-all-commit-before (i MT)
  (trace-all-commit-before i (MT-trace MT)))

(in-theory (disable MT-all-commit-before))

(defun trace-all-commit-before-trace (sub trace)
  (if (endp trace)
```

```
      t
      (if (equal sub trace)
          T
          (and (or (commit-stg-p (INST-stg (car trace)))
                   (retire-stg-p (INST-stg (car trace))))
               (trace-all-commit-before-trace sub (cdr trace))))))

(defun MT-all-commit-before-trace (trace MT)
  (trace-all-commit-before-trace trace (MT-trace MT)))

(in-theory (disable MT-all-commit-before-trace))

(encapsulate nil
(local
(defthm car-trace-INST-at-rob-head-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (in-order-rob-trace-p trace idx)
                  (not (committed-p (car sub)))
                  (subtrace-p trace MT)
                  (tail-p sub trace)
                  (INST-listp sub)
                  (uniq-inst-of-tag-in-trace idx trace)
                  (trace-all-commit-before-trace sub trace))
             (equal (inst-of-tag-in-trace idx trace)
                    (car sub)))
  :hints (("goal" :in-theory (e/d (committed-p dispatched-p)
                                  (INST-is-at-one-of-the-stages)))
          ("subgoal *1/2" :use (:instance INST-is-at-one-of-the-stages
                                          (i (car trace)))))))

; The instruction at the head of the ROB is the first uncommitted instruction
; in program order.
(defthm car-trace-INST-at-rob-head
    (implies (and (inv MT MA)
                  (MA-state-p MA) (MAETT-p MT)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (not (committed-p (car trace)))
                  (uniq-inst-of-tag (MT-rob-head MT) MT)
                  (MT-all-commit-before-trace trace MT))
             (equal  (inst-of-tag (MT-rob-head MT) MT)
                     (car trace)))
  :hints (("goal" :in-theory (enable subtrace-p MT-all-commit-before-trace
                                     inv in-order-ROB-p
                                     uniq-inst-of-tag
                                     inst-of-tag)
                  :use (:instance car-trace-INST-at-rob-head-help
                                  (sub trace) (trace (MT-trace MT))
                                  (idx (MT-rob-head MT))))))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (MA-state-p MA) (MAETT-p MT)
                           (subtrace-p trace MT)
                           (INST-listp trace)
                           (not (committed-p (car trace)))
                           (uniq-inst-of-tag (MT-rob-head MT) MT)
                           (MT-all-commit-before-trace trace MT))
                      (equal  (car trace)
                              (inst-of-tag (MT-rob-head MT) MT))))))
```

```
)

(in-theory (disable (:rewrite car-trace-INST-at-rob-head . 1)
                    (:rewrite car-trace-INST-at-rob-head . 2)))

(encapsulate nil
(local
(defthm robe-at-head-if-MT-all-commit-before-help
    (implies (and (IN-order-rob-trace-p trace rix)
                  (member-equal i trace)
                  (trace-all-commit-before i trace)
                  (not (committed-p i))
                  (dispatched-p I)
                  (INST-p i) (INST-listp trace))
             (equal (INST-tag i) rix))
  :hints (("goal" :in-theory (enable committed-p dispatched-p)))))

(defthm robe-at-head-if-MT-all-commit-before
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MT-all-commit-before i MT)
                  (not (committed-p i))
                  (dispatched-p i)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (equal (MT-rob-head MT) (INST-tag i)) t))
  :hints (("goal" :in-theory (enable INST-in inv MT-all-commit-before
                                     IN-ORDER-ROB-P))))
)

(encapsulate nil
(local
(defthm not-MT-all-commit-before-trace-nil-if-commit-inst-help-help
    (implies (uniq-inst-of-tag-in-trace rix trace)
             (not (trace-all-commit-before-trace nil trace)))))

(local
(defthm not-MT-all-commit-before-trace-nil-if-commit-inst-help
    (implies (uniq-inst-of-tag (MT-ROB-head MT) MT)
             (not (MT-all-commit-before-trace nil MT)))
  :hints (("goal" :in-theory (enable uniq-inst-of-tag
                                     MT-all-commit-before-trace)))))

; There exist uncommitted instructions in MT if commit-inst? is 1.
; Consider the relation with committed-p-car-if-MT-all-commit-before-trace.
(defthm not-MT-all-commit-before-trace-nil-if-commit-inst
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (commit-inst? MA)))
             (not (MT-all-commit-before-trace nil MT)))
  :hints (("goal" :in-theory
                  (enable incompatible-with-excpt-in-MAETT-lemmas))))
)

(encapsulate  nil
(local
(defthm not-MT-all-commit-before-trace-cdr-help
    (implies (and (tail-p sub trace)
                  (not (trace-all-commit-before-trace sub trace)))
             (not (trace-all-commit-before-trace (cdr sub) trace)))))

(defthm not-MT-all-commit-before-trace-cdr
    (implies (and (subtrace-p trace MT)
```

```
                (not (MT-all-commit-before-trace trace MT)))
           (not (MT-all-commit-before-trace (cdr trace) MT)))
  :hints (("Goal" :in-theory (enable subtrace-p MT-all-commit-before-trace))))
)

(defthm MT-all-commit-before-trace-MT-trace
    (MT-all-commit-before-trace (MT-trace MT) MT)
  :hints (("goal" :in-theory (enable MT-all-commit-before-trace))))

(encapsulate nil
(local
 (defthm MT-all-commit-before-cdr-trace-help
     (implies (and (consp sub)
                   (trace-all-commit-before-trace sub trace)
                   (committed-p (car sub)))
              (trace-all-commit-before-trace (cdr sub) trace))))

(defthm MT-all-commit-before-cdr-trace
    (implies (and (consp trace)
                  (MT-all-commit-before-trace trace MT)
                  (committed-p (car trace)))
             (MT-all-commit-before-trace (cdr trace) MT))
  :hints (("goal" :in-theory (enable MT-all-commit-before-trace))))
)

(encapsulate nil
(local
 (defthm local-help
     (implies (and (inv MT MA)
                   (subtrace-p trace MT)
                   (consp trace)
                   (consp sub)
                   (committed-p (car sub))
                   (not (committed-p (car trace)))
                   (MAETT-p MT) (MA-state-p MA))
              (not (tail-p sub (cdr trace))))
   :hints (("goal" :cases ((member-equal (car sub) (cdr trace))))
           ("subgoal 2" :in-theory
                        (disable
                         NOT-MEMBER-EQUAL-CDR-IF-CAR-IS-NOT-COMMIT)))))

(local
(defthm MT-all-commit-before-trace-if-car-is-commit-help
    (implies (and (inv MT MA)
                  (consp sub) (committed-p (car sub))
                  (subtrace-p trace MT)
                  (tail-p sub trace)
                  (MAETT-p MT) (MA-state-p MA))
             (trace-all-commit-before-trace sub trace))))

(defthm MT-all-commit-before-trace-if-car-is-commit
    (implies (and (inv MT MA)
                  (consp trace) (committed-p (car trace))
                  (subtrace-p trace MT)
                  (MAETT-p MT) (MA-state-p MA))
             (MT-all-commit-before-trace trace MT))
  :hints (("goal" :in-theory (enable MT-all-commit-before-trace subtrace-p))))
)

(encapsulate nil
(local
(defthm no-commit-inst-precedes-if-all-commit-before-help
```

```
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (trace-all-commit-before i trace)
                  (not (committed-p j)))
             (not (member-in-order j i trace)))
  :hints (("goal" :in-theory (enable member-in-order*
                                     committed-p)))))

(defthm no-commit-inst-precedes-if-all-commit-before
    (implies (and (inv MT MA)
                  (MT-all-commit-before i MT)
                  (not (committed-p j)))
             (not (inst-in-order-p j i MT)))
  :Hints (("Goal" :in-theory (enable MT-all-commit-before inst-in-order-p
                                     subtrace-p))))
)

(encapsulate nil
(local
(defthm MT-all-commit-before-trace-MT-non-commit-trace-help
    (trace-all-commit-before-trace (non-commit-trace trace) trace)))

(defthm MT-all-commit-before-trace-MT-non-commit-trace
    (MT-all-commit-before-trace (MT-non-commit-trace MT) MT)
  :hints (("goal" :in-theory (enable MT-all-commit-before-trace
                                     MT-non-commit-trace))))
)

(encapsulate nil
(local
 (defthm not-MT-all-commit-before-if-inst-specultv-help
     (implies (and (trace-correct-speculation-p trace)
                   (member-equal i trace) (INST-p i)
                   (INST-listp trace)
                   (b1p (inst-specultv? i)))
              (not (trace-all-commit-before i trace)))))

(defthm not-MT-all-commit-before-if-inst-specultv
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (b1p (inst-specultv? i))
                  (MAETT-p MT) (MA-state-p MA))
             (not (MT-all-commit-before i MT)))
  :hints (("goal" :In-theory (enable MT-all-commit-before INST-in
                                     inv correct-speculation-p))))
)

(encapsulate nil
(local
(defthm MT-all-commit-before-car-help
    (implies (and (consp sub)
                  (trace-all-commit-before-trace sub trace))
             (trace-all-commit-before (car sub) trace))))

(defthm MT-all-commit-before-car
    (implies (and (consp trace)
                  (MT-all-commit-before-trace trace MT))
             (MT-all-commit-before (car trace) MT))
  :hints (("goal" :in-theory (enable MT-all-commit-before-trace
                                     MT-all-commit-before))))
)
```

```
(encapsulate nil
(local
(defthm not-MT-all-commit-before-car-help
    (implies (and (distinct-member-p trace)
                  (tail-p sub trace)
                  (consp sub)
                  (not (trace-all-commit-before-trace sub trace)))
             (not (trace-all-commit-before (car sub) trace)))))

(defthm not-MT-all-commit-before-car
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (consp trace)
                  (not (MT-all-commit-before-trace trace MT)))
             (not (MT-all-commit-before (car trace) MT)))
  :hints (("goal" :in-theory (enable MT-all-commit-before-trace
                                     subtrace-p inv weak-inv
                                     MT-distinct-inst-p
                                     MT-all-commit-before))))
)

(encapsulate nil
(local
(defthm not-uniq-inst-of-tag-in-trace-if-no-dispatched-inst-p
    (implies (no-dispatched-inst-p trace)
             (not (uniq-inst-of-tag-in-trace rix trace)))))

(local
(defthm MT-all-commit-before-INST-at-rob-head-help
    (implies (and (INST-listp trace)
                  (in-order-trace-p trace)
                  (in-order-rob-trace-p trace rix)
                  (uniq-inst-of-tag-in-trace rix trace))
             (trace-all-commit-before (inst-of-tag-in-trace rix trace)
                                      trace))
  :hints (("goal" :in-theory (enable dispatched-p)))))

; All instructions before the instruction at the head of the ROB are
; committed.
(defthm MT-all-commit-before-INST-at-rob-head
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-of-tag (MT-ROB-head MT) MT))
             (MT-all-commit-before (inst-of-tag (MT-ROB-head MT) MT) MT))
  :hints (("goal" :in-theory (enable MT-all-commit-before
                                     inst-of-tag
                                     uniq-inst-of-tag
                                     inv
                                     in-order-dispatch-commit-p
                                     in-order-ROB-p))))
)

(defthm MT-all-commit-before-INST-at-rob-head-2
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-tag i) (MT-rob-head MT))
                  (dispatched-p i) (not (committed-p i))
                  (MAETT-p MT) (MA-state-p MA))
             (MT-all-commit-before i MT))
  :hints (("Goal" :use (:instance MT-all-commit-before-INST-at-rob-head)
                  :in-theory (disable MT-all-commit-before-INST-at-rob-head))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;End of the theory about MT-non-commit-trace and MT-non-retire-trace.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Theories about commit again.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; If i is not completed, it does not commit this cycle.
(defthm complete-stg-p-if-INST-commit
    (implies (not (complete-stg-p (INST-stg i)))
             (not (b1p (INST-commit? i MA))))
  :hints (("goal" :in-theory (enable INST-commit? lift-b-ops)))
  :rule-classes
  ((:rewrite :corollary (implies (not (complete-stg-p (INST-stg i)))
                                 (equal (INST-commit? i MA) 0))
             :hints (("goal" :in-theory (enable INST-commit? lift-b-ops))))))

;  Proof of not-INST-commit-if-earlier-inst-not-committed
;  Suppose we have two dispatched but not committed instruction i and j,
;  I and j are in ISA execution order.  J does not commit this cycle,
;  because instructions commit in-order, and I should commit first.
(encapsulate nil
; I and J are instructions in ROB. If I and J are in the ISA execution order,
; J cannot be at the head of ROB.
; Proof: If I is (MT-rob-head MT), trivial.  If they are different,
; (INST-in-order-p (MT-rob-head MT) i MT), and we get the lemma from
; transitivity of INST-in-order-p
(local
(defthm J-is-ROB-head-if-I-is-not-commit
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in-order-p i j MT)
                  (INST-p i)
                  (INST-in i MT)
                  (not (committed-p i))
                  (INST-in j MT) (INST-p j)
                  (dispatched-p j))
             (INST-in-order-p (inst-of-tag (MT-rob-head MT) MT) j MT))
  :hints (("goal" :cases ((equal (inst-of-tag (MT-rob-head MT) MT) i))
                  :restrict ((NOT-ROB-EMPTY-IF-INST-IS-EXECUTED
                              ((i i)))))
          ("subgoal 2" :cases ((dispatched-p i))))))

(local
 (defthm no-inst-of-tag-in-trace-if-member-equal
     (implies (and (no-inst-of-tag-in-trace (INST-tag j) trace)
                   (dispatched-p j)
                   (not (committed-p j)))
              (not (member-equal j trace)))
   :hints (("goal" :in-theory (enable dispatched-p committed-p)))))

(local
 (defthm not-INST-in-order-p-if-INST-tag-equal-induct
     (implies (and (uniq-inst-of-tag-in-trace (INST-tag i) trace)
                   (not (committed-p i))
                   (dispatched-p i)
                   (not (committed-p j))
                   (dispatched-p j)
                   (equal (INST-tag j) (INST-tag i)))
              (not (member-in-order i j trace)))
   :hints (("goal" :in-theory (enable member-in-order* dispatched-p
                                      committed-p)))))
```

```
(local
(defthm not-INST-in-order-p-if-INST-tag-equal
    (implies (and (uniq-inst-of-tag (INST-tag i) MT)
                  (not (committed-p i))
                  (dispatched-p i)
                  (not (committed-p j))
                  (dispatched-p j)
                  (equal (INST-tag j) (INST-tag i)))
             (not (INST-in-order-p i j MT)))
  :hints (("Goal" :in-theory (enable INST-in-order-p uniq-inst-of-tag)))))

(local
(defthm not-equal-to-MT-rob-head-if-preceded-by-inst-of-tag-head
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p j) (INST-in j MT)
                  (uniq-inst-of-tag (MT-rob-head MT) MT)
                  (INST-in-order-p (inst-of-tag (MT-rob-head MT) MT) j MT)
                  (not (committed-p j))
                  (dispatched-p j))
             (not (equal (INST-tag j) (MT-rob-head MT))))))

(defthm not-at-ROB-head-if-earlier-inst-not-commit
    (implies (and (inv MT MA)
                  (INST-in-order-p i j MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-p j)
                  (INST-in i MT) (INST-in j MT)
                  (not (committed-p i))
                  (dispatched-p j))
             (not (equal (INST-tag j) (MT-rob-head MT))))
  :Hints (("goal" :restrict ((NOT-ROB-EMPTY-IF-INST-IS-EXECUTED ((i i))))
                  :cases ((not (dispatched-p i)) (committed-p j)))))

; Instruction j does not commit in this cycle, if a preceding instruction
; i is not yet committed.
(defthm not-INST-commit-if-earlier-inst-not-committed
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in-order-p i j MT)
                  (INST-p i) (INST-in i MT)
                  (INST-p j) (INST-in j MT)
                  (not (committed-p i)))
             (equal (INST-commit? j MA) 0))
  :hints (("goal" :in-theory (enable INST-commit? lift-b-ops
                                     equal-b1p-converter
                                     bv-eqv-iff-equal))))
)
(in-theory (disable not-INST-commit-if-earlier-inst-not-committed))

; If instruction j follows i in program order, and j commits, i
; is already committed because of in-order commit.
(defthm committed-p-if-INST-commit-following
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)  (INST-in j MT) (INST-p j)
                  (not (committed-p i))
                  (b1p (INST-commit? j MA))
                  (MAETT-p MT) (MA-state-p MA))
             (not (INST-in-order-p i j MT)))
  :hints (("goal" :in-theory (e/d (committed-p INST-commit? lift-b-ops)
                                  (complete-stg-p-if-INST-commit
```

```
                                INST-is-at-one-of-the-stages))
                  :use ((:instance complete-stg-p-if-INST-commit (i j))
                        (:instance INST-is-at-one-of-the-stages)))))

(in-theory (disable committed-p-if-INST-commit-following))

; Suppose instruction i causes an exception. When i commits,
; enter-excpt? must be asserted.
(defthm not-INST-commit-if-not-enter-excpt
    (implies (and (inv MT MA)
                  (b1p (INST-excpt? i))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (enter-excpt? MA))))
             (equal (INST-commit? i MA) 0))
  :hints (("goal" :in-theory (enable INST-commit? enter-excpt? lift-b-ops
                                     bv-eqv-iff-equal
                                     inst-excpt-detected-p
                                     commit-inst?
                                     equal-b1p-converter))))

(defthm not-INST-commit-if-not-INST-excpt
    (implies (and (inv MT MA)
                  (complete-stg-p (INST-stg i))
                  (not (b1p (INST-excpt? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (b1p (enter-excpt? MA)))
             (equal (INST-commit? i MA) 0))
  :hints (("goal" :in-theory (enable INST-commit? enter-excpt? lift-b-ops
                                     bv-eqv-iff-equal
                                     inst-excpt-detected-p
                                     commit-inst?
                                     equal-b1p-converter))))

(defthm not-INST-commit-if-not-commit-jmp
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (complete-stg-p (INST-stg i))
                  (b1p (INST-wrong-branch? i))
                  (not (b1p (commit-jmp? MA)))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (equal (INST-commit? i MA) 0))
  :hints (("goal" :in-theory (e/d (lift-b-ops commit-jmp? commit-inst?
                                              equal-b1p-converter
                                              INST-excpt?
                                              INST-commit? INST-wrong-branch?)
                                  (incompatible-with-excpt-in-MAETT-lemmas)))))

(defthm not-INST-commit-if-not-leave-or-enter-excpt
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (complete-stg-p (INST-stg i))
                  (b1p (INST-context-sync? i))
                  (not (b1p (commit-jmp? MA)))
```

```
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i))))
           (equal (INST-commit? i MA) 0))
  :hints (("goal" :in-theory (e/d (lift-b-ops  commit-inst?
                                               INST-excpt?
                                               equal-b1p-converter
                                               INST-context-sync?
                                               commit-jmp? INST-commit?)
                                  (incompatible-with-excpt-in-MAETT-lemmas)))))

(defthm not-INST-commit-if-not-flush-all
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (complete-stg-p (INST-stg i))
                  (b1p (INST-start-specultv? i))
                  (not (b1p (flush-all? MA sigs))))
             (equal (INST-commit? i MA) 0))
  :hints (("goal" :in-theory (enable INST-start-specultv? lift-b-ops
                                     flush-all?))))

; If i is a speculative executed instruction, it is not at the head of ROB.
(defthm ROBE-not-head-if-inst-specultv
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (complete-stg-p (INST-stg i))
                  (b1p (inst-specultv? i))
                  (MAETT-p MT) (MA-state-p MA))
             (not (equal (INST-tag i) (MT-ROB-head MT))))
  :hints (("goal" :in-theory (disable tag-identity)
                  :use (:instance tag-identity
                                  (j (MT-1st-non-commit-inst MT))))))

(encapsulate nil
(local
(defthm inst-specultv-INST-at-ROB-head-if-commit-inst-help-help
    (implies (and (trace-correct-speculation-p trace)
                  (trace-no-specultv-commit-p trace)
                  (member-equal i trace)
                  (trace-all-commit-before i trace))
             (not (b1p (inst-specultv? i))))))

(local
(defthm inst-specultv-INST-at-ROB-head-if-commit-inst-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (MT-all-commit-before i MT))
             (not (b1p (inst-specultv? i))))
  :hints (("goal" :in-theory (enable MT-all-commit-before
                                     inv correct-speculation-p
                                     INST-in
                                     no-specultv-commit-p)))))

; The instruction at ROB head is not speculative
(defthm inst-specultv-INST-at-ROB-head-if-uniq-INST-at-ROB-head
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-of-tag (MT-rob-head MT) MT))
             (equal (inst-specultv? (inst-of-tag (MT-ROB-head MT) MT))
```

```
                  0))
  :hints (("goal" :in-theory (enable equal-b1p-converter lift-b-ops))))

)

; If commit-INST? is true, the instruction at the head of ROB commits.
(defthm INST-commit-INST-at-MT-rob-head
    (implies (and (inv MT MA)
                  (b1p (commit-INST? MA))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-commit? (inst-of-tag (MT-ROB-head MT) MT) MA) 1))
  :hints (("goal" :in-theory (e/d (INST-commit? lift-b-ops
                                                committed-p dispatched-p
                                                commit-inst?
                                                equal-b1p-converter)
                                  (INST-OF-TAG-IS-dispatched
                                   EXECUTE-OR-COMPLETE-STG-P-INST-OF-TAG
                                   INST-OF-TAG-IS-not-committed))
                  :use ((:instance INST-OF-TAG-IS-dispatched
                                   (rix (MT-rob-head MT)))
                        (:instance INST-OF-TAG-IS-not-committed
                                   (rix (MT-rob-head MT)))))))

; If i is a speculatively executed instruction, i does not commit.
(defthm not-INST-commit-if-inst-specultv
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (b1p (inst-specultv? i))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-commit? i MA) 0))
  :hints (("goal" :in-theory (enable equal-b1p-converter lift-b-ops
                                     INST-commit?))))

; INST-commit? is always off if commit-inst? is off.
(defthm not-commit-inst-if-not-commit-inst
    (implies (not (b1p (commit-inst? MA)))
             (equal (INST-commit? i MA) 0))
  :hints (("Goal" :in-theory (enable lift-b-ops INST-commit?
                                     equal-b1p-converter))))

; If i commits, it is not externally interrupted.
(defthm not-INST-exintr-now-if-inst-commit
    (implies (b1p (INST-commit? i MA))
             (equal (INST-exintr-now? i MT sigs) 0))
  :hints (("goal" :in-theory (enable INST-commit? INST-exintr-now?
                                     lift-b-ops))))

(defthm INST-cause-jmp-if-leave-excpt
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (leave-excpt? MA)))
             (equal (INST-cause-jmp? (inst-of-tag (MT-ROB-head MT) MT)
                                     MT MA sigs)
                    1))
  :hints (("goal" :in-theory (enable INST-cause-jmp? leave-excpt?
                                     lift-b-ops equal-b1p-converter))))

(defthm not-INST-cause-jmp-if-not-INST-commit
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (INST-commit? i MA))))
             (equal (INST-cause-jmp? i MT MA sigs) 0))
```

```
  :hints (("goal" :in-theory (enable INST-cause-jmp? INST-commit? lift-b-ops
                                     equal-b1p-converter))))

(defthm not-all-commit-before-if-execute-stg-p
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (execute-stg-p (INST-stg i))
                  (or (b1p (commit-jmp? MA)) (b1p (enter-excpt? MA))
                      (b1p (leave-excpt? MA)))
                  (MAETT-p MT) (MA-state-p MA))
             (not (MT-all-commit-before i MT)))
  :hints (("goal" :use (:instance robe-at-head-if-MT-all-commit-before)
                  :in-theory (e/d (commit-jmp? enter-excpt?
                                               leave-excpt? lift-b-ops)
                                  (robe-at-head-if-MT-all-commit-before)))))

(defthm INST-cause-jmp-if-complete-stg-p
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (complete-stg-p (INST-stg i))
                  (MT-all-commit-before i MT)
                  (or (b1p (commit-jmp? MA))
                      (b1p (enter-excpt? MA))
                      (b1p (leave-excpt? MA)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-cause-jmp? i MT MA sigs) 1))
  :hints (("goal" :use (:instance robe-at-head-if-MT-all-commit-before)
                  :in-theory (e/d (INST-cause-jmp? lift-b-ops
                                                   equal-b1p-converter)
                                  (robe-at-head-if-MT-all-commit-before)))))

(encapsulate nil
(local
 (defthm not-subtrace-after-p-if-out-of-order-dispatch
     (implies (and (inv MT MA)
                   (INST-in i MT)
                   (consp trace)
                   (dispatched-p (car trace))
                   (not (dispatched-p i))
                   (MAETT-p MT) (MA-state-p MA))
              (not (subtrace-after-p i trace MT)))
   :hints (("goal" :use (:instance inst-in-order-dispatched-undispatched
                                   (i (car trace)) (j i))
                   :in-theory (disable
                               INST-IN-ORDER-DISPATCHED-UNDISPATCHED)))))

(local
(defthm no-inst-at-MT-rob-head-if-MT-all-commit-before-help-help
    (implies (and (inv MT MA)
                  (not (dispatched-p i))
                  (INST-in i MT)
                  (subtrace-after-p i trace MT)
                  (MAETT-p MT) (MA-state-p MA))
             (no-inst-of-tag-in-trace rix trace))))

(local
(defthm no-inst-at-MT-rob-head-if-MT-all-commit-before-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (member-equal i trace) (INST-p i)
                  (trace-all-commit-before i trace)
                  (not (dispatched-p i))
```

```
                (MAETT-p MT) (MA-state-p MA))
           (no-inst-of-tag-in-trace rix trace))))

(defthm no-inst-at-MT-rob-head-if-MT-all-commit-before
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (MT-all-commit-before i MT)
                (not (dispatched-p i))
                (MAETT-p MT) (MA-state-p MA))
           (no-inst-of-tag rix MT))
  :hints (("goal" :in-theory (enable no-inst-of-tag INST-in
                                     MT-all-commit-before))))
)

(defthm not-MT-all-commit-before-undispatched
    (implies (and (inv MT MA)
                (or (DQ-stg-p (INST-stg i))
                    (IFU-stg-p (INST-stg i)))
                (not (b1p (rob-empty? (MA-rob MA))))
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA))
           (not (MT-all-commit-before i MT)))
  :hints (("Goal" :use (:instance
                        no-inst-at-MT-rob-head-if-MT-all-commit-before
                        (rix (MT-ROB-head MT)))
                :in-theory
                (disable NO-INST-AT-MT-ROB-HEAD-IF-MT-ALL-COMMIT-BEFORE))))

(defthm INST-cause-jmp-if-commit-jmp
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (MT-all-commit-before i MT)
                (not (committed-p i))
                (or (b1p (commit-jmp? MA))
                    (b1p (enter-excpt? MA))
                    (b1p (leave-excpt? MA)))
                (MAETT-p MT) (MA-state-p MA))
           (equal (INST-cause-jmp? i MT MA sigs) 1))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                :in-theory
                (e/d (commit-jmp? leave-excpt? enter-excpt? lift-b-ops)
                     (incompatible-with-excpt-in-MAETT-lemmas
                      INST-is-at-one-of-the-stages
                      no-inst-at-MT-rob-head-if-MT-all-commit-before)))
          (when-found (IFU-stg-p (INST-stg i))
                      (:use (:instance
                             no-inst-at-MT-rob-head-if-MT-all-commit-before
                             (rix (MT-rob-head MT)))))))

(defthm INST-cause-jmp-or-INST-exintr-now-if-first-uncommit
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (MT-all-commit-before i MT)
                (not (committed-p i))
                (b1p (flush-all? MA sigs))
                (not (b1p (INST-exintr-now? i MA sigs)))
                (MAETT-p MT) (MA-state-p MA))
           (equal (INST-cause-jmp? i MT MA sigs) 1))
  :Hints (("goal" :in-theory (enable flush-all? lift-b-ops))))

(defthm committed-p-step-inst-if-INST-cause-jmp
    (implies (and (inv MT MA)
```

```
                (INST-p i) (MAETT-p MT) (MA-state-p MA)
                (b1p (INST-cause-jmp? i MT MA sigs))
                (not (commit-stg-p (INST-stg (step-INST i MT MA sigs)))))
           (retire-stg-p (INST-stg (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (enable INST-cause-jmp? lift-b-ops
                                     INST-commit?
                                     step-INST step-INST-complete))))

(in-theory (disable no-inst-at-MT-rob-head-if-MT-all-commit-before))

(defthm inst-cause-jmp-if-flush-all
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (complete-stg-p (INST-stg i))
                (MT-all-commit-before i MT)
                (b1p (flush-all? MA sigs))
                (MAETT-p MT) (MA-state-p MA))
           (equal (INST-cause-jmp? i MT MA sigs) 1))
  :hints (("Goal" :in-theory (enable INST-exintr-now? lift-b-ops))))

(defthm inst-exintr-now-if-flush-all
    (implies (and (inv MT MA)
                (or (DQ-stg-p (INST-stg i))
                    (IFU-stg-p (INST-stg i)))
                (MT-all-commit-before i MT)
                (b1p (flush-all? MA sigs))
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA))
           (equal (INST-exintr-now? i MA sigs) 1))
  :hints (("Goal" :in-theory (e/d (flush-all? lift-b-ops)
                                  (not-MT-all-commit-before-undispatched
                                   NOT-COMMIT-INST-IF-ROB-EMPTY))
                :use ((:instance not-MT-all-commit-before-undispatched)
                      (:instance NOT-COMMIT-INST-IF-ROB-EMPTY)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Lemmas about INST-start-speculative and MT-specultv
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defthm inst-start-specultv-step-INST-if-DQ-or-execute
    (implies (and (INST-p i) (MA-state-p MA) (MAETT-p MT) (MA-input-p sigs)
                (or (DQ-stg-p (INST-stg i))
                    (execute-stg-p (INST-stg i))))
           (equal (inst-start-specultv? (step-INST i MT MA sigs))
                  (inst-start-specultv? i)))
  :hints (("Goal" :in-theory (enable inst-start-specultv?))))

(defthm INST-start-specultv-step-INST-if-IFU
    (implies (and (INST-p i) (MA-state-p MA) (MAETT-p MT)
                (MA-input-p sigs) (IFU-stg-p (INST-stg i))
                (or (not (b1p (IFU-branch-predict? (MA-IFU MA) MA sigs)))
                    (b1p (DQ-full? (MA-DQ MA)))))
           (equal (INST-start-specultv? (step-INST i MT MA sigs))
                  (INST-start-specultv? i)))
  :Hints (("goal" :in-theory (enable INST-start-specultv?
                                     committed-p
                                     lift-b-ops))))

(defthm INST-start-specultv-step-INST-if-complete
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i) (MA-state-p MA) (MAETT-p MT)
                (MA-input-p sigs)
                (complete-stg-p (INST-stg i))
```

```
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (flush-all? MA sigs))))
           (equal (INST-start-specultv? (step-INST i MT MA sigs))
                  (INST-start-specultv? i)))
  :hints (("goal" :in-theory (enable equal-b1p-converter
                                     INST-start-specultv? lift-b-ops))))

(defthm INST-start-specultv-step-INST
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i) (MA-state-p MA) (MAETT-p MT)
                (MA-input-p sigs)
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (or (not (b1p (IFU-branch-predict? (MA-IFU MA) MA sigs)))
                    (b1p (DQ-full? (MA-DQ MA))))
                (not (b1p (flush-all? MA sigs))))
           (equal (INST-start-specultv? (step-INST i MT MA sigs))
                  (INST-start-specultv? i)))
  :hints (("goal" :use (:instance INST-is-at-one-of-the-stages)
                :in-theory (disable INST-is-at-one-of-the-stages))))

(defthm INST-start-specultv-step-INST-2
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (not (b1p (flush-all? MA sigs)))
                (b1p (fetch-inst? MA sigs))
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (MA-input-p sigs)
                (MA-state-p MA) (MAETT-p MT))
           (equal (INST-start-specultv? (step-INST i MT MA sigs))
                  (INST-start-specultv? i)))
  :hints (("goal" :in-theory (enable fetch-inst? lift-b-ops))))

(defthm inst-start-specultv-step-INST-if-not-retire-after-step
    (implies (and (INST-p i)
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                (complete-stg-p (INST-stg i))
                (not (commit-stg-p (INST-stg (step-INST i MT MA sigs))))
                (not (retire-stg-p (INST-stg (step-INST i MT MA sigs)))))
           (equal (inst-start-specultv? (step-INST i MT MA sigs))
                  (inst-start-specultv? i)))
  :hints (("Goal" :in-theory (enable inst-start-specultv? lift-b-ops))))

(encapsulate nil
(local
(defthm MT-inst-specultv-if-INST-start-specultv-help
    (implies (and (member-equal i trace)
                (b1p (INST-start-specultv? i)))
           (equal (trace-specultv? trace) 1))
  :hints (("goal" :in-theory (enable lift-b-ops equal-b1p-converter)))))

(defthm MT-inst-specultv-if-INST-start-specultv
    (implies (and (INST-in i MT) (b1p (INST-start-specultv? i)))
           (equal (MT-specultv? MT) 1))
  :hints (("goal" :in-theory (enable MT-specultv? INST-in))))

)

(encapsulate nil
(local
```

```
(defthm MT-specultv-MT-step-if-INST-specultv-help
    (implies (and (b1p (inst-specultv? i))
                  (member-equal i trace)
                  (not (b1p (flush-all? MA sigs))))
             (equal (trace-specultv? (step-trace trace MT MA sigs
                                                 ISA spc smc))
                    1))
  :hints (("goal" :in-theory (enable flush-all? lift-b-ops
                                     equal-b1p-converter
                                     INST-cause-jmp? INST-exintr-now?
                                     ex-intr?)))))

(defthm MT-specultv-MT-step-if-INST-specultv
    (implies (and (b1p (inst-specultv? i))
                  (INST-in i MT)
                  (not (b1p (flush-all? MA sigs))))
             (equal (MT-specultv? (MT-step MT MA sigs)) 1))
  :hints (("Goal" :in-theory (enable MT-specultv? MT-step
                                     INST-in))))
)

(defthm not-commit-stg-step-inst-if-inst-specultv
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (b1p (inst-specultv? i))
                  (not (committed-p i))
                  (MA-state-p MA) (MAETT-p MT) (MA-input-p sigs))
             (not (commit-stg-p (INST-stg (step-inst i MT MA sigs)))))
  :hints (("Goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (disable INST-is-at-one-of-the-stages))))

(defthm not-retire-stg-step-inst-if-inst-specultv
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (inst-specultv? i)))
             (not (retire-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("Goal" :use ((:instance stages-reachable-to-retire-stg))
                  :in-theory (enable NOT-INST-SPECULTV-INST-IN-IF-COMMITTED))))

(defthm not-committed-p-step-inst-if-inst-specultv
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (committed-p i))
                  (b1p (inst-specultv? i))
                  (MA-state-p MA) (MAETT-p MT) (MA-input-p sigs))
             (not (committed-p (step-inst i MT MA sigs))))
  :hints (("Goal" :in-theory (enable committed-p))))

(encapsulate nil
; If a complete instruction i causes an exception, i does not advance to
; commit stage.
(local
(defthm not-commit-stg-p-step-INST-if-inst-excpt
    (implies (and (inv MT MA)
                  (complete-stg-p (INST-stg i))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (b1p (INST-excpt? I)))
             (not (commit-stg-p (INST-stg (step-INST i MT MA sigs)))))
```

```
  :hints (("goal" :in-theory (enable step-inst-opener step-inst-complete
                                     lift-b-ops)))))

(local
(defthm not-commit-stg-p-step-INST-if-stg-is-complete
    (implies (and (INST-p I)
                  (equal (INST-stg i) '(complete)))
             (not (commit-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :Hints (("Goal" :in-theory (enable MT-def)))))

(local
(defthm not-commit-stg-p-step-INST-if-inst-wrong-branch
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (complete-stg-p (INST-stg i))
                  (b1p (INST-wrong-branch? I)))
             (not (commit-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("Goal" :cases ((b1p (INST-BU? i)))
                  :in-theory (enable complete-stg-p)))))

(local
(defthm not-commit-stg-p-step-INST-if-inst-context-sync
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p I) (INST-in I MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (complete-stg-p (INST-stg i))
                  (b1p (INST-context-sync? I)))
             (not (commit-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("goal" :in-theory (enable complete-stg-p INST-context-sync?
                                     lift-b-ops)))))

; A completed instruction which starts speculative execution does not
; advance to the commit stage.  A store instruction which is committed
; but whose memory write is not completed can be in the commit stage.
(defthm not-commit-stg-p-step-INST-if-inst-start-specultv
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p I) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (complete-stg-p (INST-stg i))
                  (b1p (inst-start-specultv? i))                    )
             (not (commit-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("Goal" :in-theory (enable inst-start-specultv? lift-b-ops))))
)

(encapsulate nil
; Instruction i does not cause a jump in this cycle, and i is an
; exception causing instruction (not an ordinary memory write operation),
; then i does not advance to retire stage from the complete stage.
; (because jumping to the exception vector is detected by INST-cause-jmp?.)
(local
(defthm not-retire-without-INST-cause-jmp-if-INST-excpt
    (implies (and (inv MT MA)
                  (MA-state-p MA) (MAETT-p MT)
                  (INST-p i)
```

```
                (INST-in i MT)
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (complete-stg-p (INST-stg i))
                (b1p (INST-excpt? i))
                (not (b1p (INST-cause-jmp? i MT MA sigs))))
           (not (retire-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("goal" :in-theory (enable step-INST-opener
                                     step-INST-complete
                                     inst-commit?
                                     enter-excpt?
                                     commit-inst?
                                     complete-stg-p
                                     INST-cause-jmp?
                                     lift-b-ops)))))

; Similar lemma for the case where instruction i is a mispredicted branch.
(local
(defthm not-retire-without-INST-cause-jmp-if-INST-wrong-branch
    (implies (and (inv MT MA)
                  (MA-state-p MA) (MAETT-p MT)
                  (INST-p i)
                  (INST-in i MT)
                  (complete-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (b1p (INST-wrong-branch? i))
                  (not (b1p (INST-cause-jmp? i MT MA sigs))))
             (not (retire-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("goal" :in-theory (enable step-INST-opener
                                     step-INST-complete
                                     inst-commit?
                                     enter-excpt?
                                     inst-wrong-branch?
                                     commit-jmp?
                                     commit-inst?
                                     complete-stg-p
                                     exception-relations
                                     INST-cause-jmp?
                                     lift-b-ops)))))

; Lemma for the case where instruction i is an context synchronization
; instruction.
(local
(defthm not-retire-without-INST-cause-jmp-if-INST-context-sync
    (implies (and (inv MT MA)
                  (MA-state-p MA) (MAETT-p MT)
                  (INST-p i)
                  (INST-in i MT)
                  (complete-stg-p (INST-stg i))
                  (b1p (INST-context-sync? i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (INST-cause-jmp? i MT MA sigs))))
             (not (retire-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("goal" :in-theory
                  (enable step-INST-opener
                          step-INST-complete
                          inst-commit?
                          inst-context-sync?
                          INST-excpt?
                          commit-jmp?
```

```
                        commit-inst?
                        complete-stg-p
                        exception-relations
                        INST-fetch-error-detected-p-iff-INST-fetch-error?
                        INST-decode-error-detected-p-iff-INST-decode-error?
                        INST-cause-jmp?
                        lift-b-ops)))))

(defthm not-retire-stg-p-step-INST-if-not-INST-cause-jmp?
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p I) (INST-in i MT)
                  (complete-stg-p (INST-stg i))
                  (b1p (inst-start-specultv? i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (INST-cause-jmp? i MT MA sigs))))
             (not (retire-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("Goal" :in-theory (enable inst-start-specultv? lift-b-ops))))
)

(defthm not-committed-p-step-INST-if-not-INST-cause-jmp?
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (INST-p I) (INST-in i MT)
                  (complete-stg-p (INST-stg i))
                  (b1p (inst-start-specultv? i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (INST-cause-jmp? i MT MA sigs))))
             (not (committed-p (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (enable committed-p))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Lemmas about MT-self-modify
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(encapsulate nil
(local
(defthm MT-self-modify-if-INST-in-modified-help
    (implies (and (b1p (INST-modified? i))
                  (member-equal i trace))
             (equal (trace-self-modify? trace) 1))
  :hints (("goal" :in-theory (enable lift-b-ops equal-b1p-converter)))))

(defthm MT-self-modify-if-INST-in-modified
    (implies (and (b1p (INST-modified? i))
                  (INST-in i MT))
             (equal (MT-self-modify? MT) 1))
  :Hints (("goal" :in-theory (enable MT-self-modify? INST-in))))
)
(defthm  INST-modified-car-if-not-MT-self-modify
    (implies (and (inv MT MA)
                  (not (b1p (MT-self-modify? MT)))
                  (subtrace-p trace MT)
                  (INST-listp trace) (consp trace)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-modified? (car trace)) 0))
  :hints (("goal" :in-theory (enable equal-b1p-converter))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  Lemmas about MT-CMI-p
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(encapsulate nil
(local
(defthm MT-CMI-p-if-INST-modified-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (b1p (INST-modified? i))
                  (committed-p i)
                  (member-equal i trace)
                  (INST-p i) (INST-listp trace) (MAETT-p MT) (MA-state-p MA))
             (trace-CMI-p trace))))

(defthm MT-CMI-p-if-INST-modified
    (implies (and (inv MT MA)
                  (b1p (INST-modified? i))
                  (committed-p i)
                  (INST-in i MT) (MAETT-p MT) (MA-state-p MA))
             (MT-CMI-p MT))
  :hints (("goal" :in-theory (enable INST-in MT-CMI-p))))
)

(encapsulate nil
(local
 (defthm MT-CMI-p-MT-step-help
     (implies (and (MA-state-p MA)
                   (MA-input-p sigs)
                   (MAETT-p MT)
                   (INST-listp trace)
                   (subtrace-p trace MT)
                   (trace-CMI-p trace))
              (trace-CMI-p (step-trace trace MT MA sigs
                                       pre spc smc)))
   :hints (("Goal" :in-theory (enable INST-EXINTR-NOW? INST-cause-jmp?
                                      lift-b-ops))))
 )

(defthm MT-CMI-p-MT-step
    (implies (and (MT-CMI-p MT)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (MA-input-p sigs))
             (MT-CMI-p (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable MT-CMI-p MT-step))))
)

(defthm commit-self-modified-p-MT-stepn
    (implies (and (MT-CMI-p MT)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (MA-input-listp sigs-lst)
                  (<= n (len sigs-lst)))
             (MT-CMI-p (MT-stepn MT MA sigs-lst n))))

(defthm INST-exintr-now-INST-commit-exclusive
    (implies (and (inv MT MA)
                  (b1p (INST-commit? i MA))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-exintr-now? j MA sigs) 0))
  :hints (("goal" :in-theory (enable INST-exintr-now? INST-commit?
                                     lift-b-ops equal-b1p-converter)
                  :restrict ((NOT-ROB-EMPTY-IF-INST-IS-EXECUTED
```

```
                              ((i i)))))))
(in-theory (disable INST-exintr-now-INST-commit-exclusive))

(encapsulate nil

(local
(defthm not-INST-cause-jmp-if-another-INST-commit
    (implies (and (inv MT MA)
                  (b1p (INST-commit? i MA))
                  (INST-in i MT) (INST-p i) (INST-in j MT) (INST-p j)
                  (not (equal i j))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-cause-jmp? j MT MA sigs) 0))
  :hints (("goal" :in-theory (enable INST-cause-jmp? INST-commit?
                                     lift-b-ops equal-b1p-converter)))))

(local
 (defthm MT-CMI-p-if-modified-INST-commit-help
     (implies (and (inv MT MA)
                   (b1p (INST-commit? j MA))
                   (INST-in-order-p i j MT)
                   (not (commit-stg-p (INST-stg (step-INST i MT MA sigs))))
                   (INST-in i MT) (INST-p i) (INST-in j MT) (INST-p j)
                   (MAETT-p MT) (MA-state-p MA))
              (retire-stg-p (INST-stg (step-INST i MT MA sigs))))
   :hints (("goal" :cases ((committed-p i))
                   :in-theory (enable committed-p inst-commit?
                                      lift-b-ops)))))

(local
(defthm MT-CMI-p-if-modified-INST-commit-induct
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (member-equal i trace) (INST-p i)
                  (INST-listp trace)
                  (b1p (INST-modified? i))
                  (b1p (INST-commit? i MA))
                  (MAETT-p MT) (MA-state-p MA))
             (trace-CMI-p (step-trace trace MT MA sigs
                                      isa spc smc)))
  :hints (("goal" :in-theory
                  (enable NOT-INST-COMMIT-IF-EARLIER-INST-NOT-COMMITTED
                          INST-exintr-now-INST-commit-exclusive
                          committed-p)
                  :restrict ((NOT-INST-COMMIT-IF-EARLIER-INST-NOT-COMMITTED
                              ((i (car trace)))))))))

(defthm MT-CMI-p-if-modified-INST-commit
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (b1p (INST-modified? i))
                  (b1p (INST-commit? i MA))
                  (MAETT-p MT) (MA-state-p MA))
             (MT-CMI-p (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable MT-CMI-p
                                     INST-in
                                     MT-step))))
)

(defthm not-retire-stg-step-inst-if-INST-modified
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
```

```
                (MAETT-p MT) (MA-state-p MA)
                (MA-input-p sigs)
                (not (MT-CMI-p (MT-step MT MA sigs)))
                (b1p (INST-modified? i)))
           (not (retire-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("Goal" :use ((:instance stages-reachable-to-retire-stg))
                  :in-theory (enable inst-stg-step-inst lift-b-ops))))
;;;;;
;;;;Misc lemmas about inst-exintr-now?
(defthm IFU-or-DQ-stg-if-inst-exintr-now
    (implies (and (not (IFU-stg-p (INST-stg i)))
                  (not (DQ-stg-p (INST-stg i))))
             (not (b1p (inst-exintr-now? i MA sigs))))
  :hints (("Goal" :in-theory (enable inst-exintr-now? lift-b-ops))))

(defthm inst-cause-jmp-inst-exintr-now-exclusive
    (implies (and (INST-p i) (b1p (inst-cause-jmp? i MT MA sigs)))
             (not (b1p (inst-exintr-now? i MA sigs))))
  :hints (("goal" :use (:instance INST-is-at-one-of-the-stages)
                  :in-theory (disable INST-is-at-one-of-the-stages))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MT-specultv-at-dispatch?
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(encapsulate nil
(local
(defthm local-help1
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (not (dispatched-p i))
                  (member-equal i trace) (INST-p i)
                  (not (b1p (inst-specultv? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (not (b1p (trace-specultv-at-dispatch? trace))))
  :hints (("goal" :in-theory (enable INST-START-SPECULTV-CAR)))))

; If a non-dispatched instruction is not speculatively executed,
; currently no dispatched instructions are speculatively executed.
(defthm MT-specultv-at-dispatch-off-if-non-specultv-inst-in
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (dispatched-p i))
                  (not (b1p (inst-specultv? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (MT-specultv-at-dispatch? MT) 0))
  :hints (("Goal" :in-theory (enable MT-specultv-at-dispatch?
                                     equal-b1p-converter INST-in))))
)
(in-theory (disable MT-specultv-at-dispatch-off-if-non-specultv-inst-in))

(encapsulate nil
(local
(defthm local-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (member-equal i trace) (INST-p i)
                  (not (dispatched-p i))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (trace-modified-at-dispatch? trace) 0))))

(defthm INST-modified-at-dispatch-off-if-undispatched-inst-in
```

```
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (dispatched-p i))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (MT-modified-at-dispatch? MT) 0))
  :hints (("Goal" :in-theory (enable MT-modified-at-dispatch? INST-in))))
)
(in-theory (disable INST-modified-at-dispatch-off-if-undispatched-inst-in))

; If an instruction i is dispatched, but not yet committed, it is not
; speculatively executed, and if flush-all?  is not true, then the
; value of INST-start-specultv?  won't change.  If the instruction is
; at the execute stage, this is true from the definition of
; INST-start-specultv?.  If it is at complete-stg-p, the instruction
; will not retire because flush-all? is 0.
(defthm INST-start-specultv-if-not-flush-all
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (dispatched-p i)
                  (not (committed-p i))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (inst-specultv? i)))
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (equal (INST-start-specultv? (step-INST i MT MA sigs))
                    (INST-start-specultv? i)))
  :hints (("Goal" :in-theory (e/d (dispatched-p committed-p
                                   equal-b1p-converter)
                                  (inst-is-at-one-of-the-stages))
                  :cases ((b1p (INST-modified? i))))
          ("subgoal 1" :cases ((committed-p (step-INST i MT MA sigs))))
          ("subgoal 1.1" :in-theory (e/d (dispatched-p committed-p
                                          step-inst-low-level-functions
                                          step-inst-complete-inst
                                          lift-b-ops
                                          equal-b1p-converter
                                   step-inst-complete-inst)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm MT-specultv-at-dispatch-MT-step-help-help
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (committed-p i))
                  (b1p (INST-start-specultv? i))
                  (not (b1p (flush-all? MA sigs)))
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (not (committed-p (step-INST i MT MA sigs))))
  :hints (("Goal" :cases ((b1p (inst-specultv? i))
                          (b1p (INST-modified? i)))
                  :in-theory (e/d ( step-inst-complete-inst
                                    step-inst-low-level-functions
                                    lift-b-ops
                                    committed-p*)
                                  (INST-IS-AT-ONE-OF-THE-STAGES))))))

(local
(defthm MT-specultv-at-dispatch-MT-step-help
    (implies (and (inv MT MA)
```

```
                (subtrace-p trace MT) (INST-listp trace)
                (not (MT-CMI-p (MT-step MT MA sigs)))
                (b1p (trace-specultv-at-dispatch? trace))
                (not (b1p (flush-all? MA sigs)))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (equal (trace-specultv-at-dispatch?
                    (step-trace trace MT MA sigs ISA spc smc))
                  1))
  :hints (("Goal" :in-theory (e/d (lift-b-ops)
                                  (INST-IS-AT-ONE-OF-THE-STAGES))))))

; If we are dispatching instructions speculatively in the current cycle,
; and if flush-all? is not true, we are still dispatching speculatively
; in the next cycle.
(defthm MT-specultv-at-dispatch-MT-step
    (implies (and (inv MT MA)
                (b1p (MT-specultv-at-dispatch? MT))
                (not (MT-CMI-p (MT-step MT MA sigs)))
                (not (b1p (flush-all? MA sigs)))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (equal (MT-specultv-at-dispatch? (MT-step MT MA sigs)) 1))
  :hints (("Goal" :in-theory (enable MT-specultv-at-dispatch?))))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MT-no-jmp-exintr-before
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; We define a predicate to check whether instructions are abandoned by
; branching, exceptions, and other context synchronizing events.
; (MT-no-jmp-exintr-before i MT MA sigs) returns nil if a committing
; instruction which appears earlier than instruction i in MT is either
; a mispredicted branch, an external or internal exception, or other
; instructions that makes machine synchronize.  For instance, return
; from exception handling.
; (MT-no-jmp-exintr-before-trace sub MT MA sigs) checks if there appears
; such an instruction before a terminating list, sub, of MAETT, MT.
; Lemma MT-no-jmp-exintr-before-INST-and-trace shows the relation of
; these two predicates.

(defun trace-no-jmp-exintr-before (i trace MT MA sigs)
  (declare (xargs :guard (and (INST-p i) (INST-listp trace)
                              (MAETT-p MT) (MA-state-p MA)
                              (MA-input-p sigs))))
  (if (endp trace)
      t
      (if (equal i (car trace))
          t
          (if (or (b1p (INST-cause-jmp? (car trace) MT MA sigs))
                  (b1p (INST-exintr-now? (car trace) MA sigs)))
              nil
              (trace-no-jmp-exintr-before i (cdr trace) MT MA sigs)))))

(defun MT-no-jmp-exintr-before (i MT MA sigs)
  (declare (xargs :guard (and (INST-p i) (MAETT-p MT) (MA-state-p MA)
                              (MA-input-p sigs))))
  (trace-no-jmp-exintr-before i (MT-trace MT) MT MA sigs))

(in-theory (disable MT-no-jmp-exintr-before))

(encapsulate nil
(local
(defthm MT-no-jmp-exintr-before-cadr-help
```

```
    (implies (and (consp trace)
                  (consp (cdr trace))
                  (distinct-member-p trace2)
                  (not (b1p (INST-cause-jmp? (car trace) MT MA sigs)))
                  (not (b1p (INST-exintr-now? (car trace) MA sigs)))
                  (tail-p trace trace2)
                  (trace-no-jmp-exintr-before (car trace) trace2
                                                        MT MA sigs))
             (trace-no-jmp-exintr-before (cadr trace) trace2 MT MA sigs))
  :hints (("goal" :in-theory (enable MT-no-jmp-exintr-before)))))

(defthm MT-no-jmp-exintr-before-cadr
    (implies (and (weak-inv MT)
                  (consp trace)
                  (consp (cdr trace))
                  (not (b1p (INST-cause-jmp? (car trace) MT MA sigs)))
                  (not (b1p (INST-exintr-now? (car trace) MA sigs)))
                  (subtrace-p trace MT)
                  (MT-no-jmp-exintr-before (car trace) MT MA sigs))
             (MT-no-jmp-exintr-before (cadr trace) MT MA sigs))
  :hints (("goal" :in-theory (enable MT-no-jmp-exintr-before subtrace-p
                                     weak-inv MT-distinct-INST-p
                                     ))))
)

(defthm MT-no-jmp-exintr-before-car-MT-trace
    (MT-no-jmp-exintr-before (car (MT-trace MT)) MT MA sigs)
  :hints (("goal" :in-theory (enable MT-no-jmp-exintr-before))))

(defun trace-no-jmp-exintr-before-trace (sub trace MT MA sigs)
  (declare (xargs :guard (and (INST-listp sub) (INST-listp trace) (MAETT-p MT)
                              (MA-state-p MA) (MA-input-p sigs))))
  (if (endp trace)
      t
      (if (equal sub trace)
          t
          (if (or (b1p (INST-cause-jmp? (car trace) MT MA sigs))
                  (b1p (INST-exintr-now? (car trace) MA sigs)))
              nil
              (trace-no-jmp-exintr-before-trace sub (cdr trace) MT MA sigs)))))

(defun MT-no-jmp-exintr-before-trace (trace MT MA sigs)
  (declare (xargs :guard (and (INST-listp trace) (MAETT-p MT)
                              (MA-state-p MA) (MA-input-p sigs))))
  (trace-no-jmp-exintr-before-trace trace (MT-trace MT) MT MA sigs))

(in-theory (disable MT-no-jmp-exintr-before-trace))

(encapsulate nil
(local
(defthm MT-no-jmp-exintr-before-cdr-help
    (implies (and (consp trace)
                  (distinct-member-p trace2)
                  (not (b1p (INST-cause-jmp? (car trace) MT MA sigs)))
                  (not (b1p (INST-exintr-now? (car trace) MA sigs)))
                  (tail-p trace trace2)
                  (trace-no-jmp-exintr-before-trace trace trace2 MT MA sigs))
             (trace-no-jmp-exintr-before-trace (cdr trace) trace2 MT MA sigs))
  :hints (("goal" :in-theory (enable MT-no-jmp-exintr-before-trace)))))

(defthm MT-no-jmp-exintr-before-cdr
    (implies (and (weak-inv MT)
```

```
                (consp trace)
                (not (b1p (INST-cause-jmp? (car trace) MT MA sigs)))
                (not (b1p (INST-exintr-now? (car trace) MA sigs)))
                (subtrace-p trace MT)
                (MT-no-jmp-exintr-before-trace trace MT MA sigs))
           (MT-no-jmp-exintr-before-trace (cdr trace) MT MA sigs))
  :hints (("goal" :in-theory (enable MT-no-jmp-exintr-before-trace subtrace-p
                                     weak-inv MT-distinct-INST-p))))
)

(defthm MT-no-jmp-exintr-before-trace-MT-trace
    (MT-no-jmp-exintr-before-trace (MT-trace MT) MT MA sigs)
  :hints (("goal" :in-theory (enable MT-no-jmp-exintr-before-trace))))

(defthm MT-no-jmp-exintr-before-INST-and-trace-help
    (implies (and (distinct-member-p trace)
                  (tail-p sub trace)
                  (consp sub))
             (iff (trace-no-jmp-exintr-before (car sub) trace MT MA sigs)
                  (trace-no-jmp-exintr-before-trace sub trace MT MA sigs))))

(defthm MT-no-jmp-exintr-before-INST-and-trace
    (implies (and (weak-inv MT)
                  (subtrace-p trace MT)
                  (consp trace))
             (iff (MT-no-jmp-exintr-before (car trace) MT MA sigs)
                  (MT-no-jmp-exintr-before-trace trace MT MA sigs)))
  :hints (("goal" :in-theory (enable MT-no-jmp-exintr-before
                                     MT-no-jmp-exintr-before-trace
                                     weak-inv
                                     subtrace-p
                                     MT-distinct-INST-p))))


(deftheory INST-exunit-relations
    '(INST-sync-INST-BU-exclusive-2      NOT-INST-RFEH-IF-INST-LSU?
      NOT-INST-WB-SREG-IF-INST-LSU?      NOT-INST-SYNC-IF-INST-LSU?
      INST-writeback-p-INST-BU-exclusive))

(in-theory (disable INST-exunit-relations))
```

D.5.4 MAETT-lemmas.lisp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MAETT-lemmas.lisp
; Author  Jun Sawada, University of Texas at Austin
;
;  This book combines the results of MAETT-lemmas1.lisp and
;  and MAETT-lemmas2.lisp.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(in-package "ACL2")

(include-book "MA2-lemmas")
(include-book "invariants-def")

(deflabel begin-MAETT-lemmas)
(include-book "MAETT-lemmas1")
(include-book "MAETT-lemmas2")
```

D.6 Invariant Proofs

This section proves that `inv` is in fact an invariant. This requires to prove that all properties used in the definition of `inv` are true in the next MA state, given that `inv` is true in the current state. Most of the properties are defined as a predicate of both the MA state and its MAETT.

The first two files memory-inv.tex and modifier.tex proves lemmas used in the following proofs. Actually, they can be categorized into the layer for shared lemmas.

All of the following files except invariant-proof.lisp prove individual properties independently. These results are combined together to the invariant proof in invariant-proof.lisp. The result proved in this section is used in the proof of the correctness in the next section.

D.6.1 memory-inv.lisp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MI-inv.lisp
; Author  Jun Sawada, University of Texas at Austin
;
;  This book proves basic lemmas about the memory access and its
;  protection.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(in-package "ACL2")

(include-book "MA2-lemmas")
(include-book "MAETT-lemmas")

(deflabel begin-memory-inv)
; This book proves basic lemmas about the memory.
;  Index
;    Lemmas about memory protection
;    Lemmas about supervisor/user mode
;    Lemmas about side effects of memory operations

(local
(defthm MT-mem-=-MA-mem
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA))
             (equal (MT-mem MT) (MA-mem MA)))
  :hints (("goal" :in-theory (enable weak-inv inv
                                     mem-match-p)))))

(local
(defthm MT-RF-=-MA-RF
```

```
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA))
             (equal (MT-RF MT) (MA-RF MA)))
  :hints (("goal" :in-theory (enable weak-inv inv
                                     RF-match-p)))))

(local
(defthm MT-SRF-=-MA-SRF
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA))
             (equal (MT-SRF MT) (MA-SRF MA)))
  :hints (("goal" :in-theory (enable weak-inv inv
                                     SRF-match-p)))))

(local (in-theory (enable ISA-before-MT-non-nil-trace)))

;; Lemmas about memory protection.

; IFU-fetch-prohibited? and read-error? are equivalent.
(defthm IFU-fetch-prohibited-read-error
    (equal (IFU-fetch-prohibited? addr mem su)
           (read-error? addr mem su))
  :hints (("goal" :in-theory (enable IFU-fetch-prohibited? read-error?))))

; Read protection does not change after ISA-step.
(defthm readable-addr-ISA-step
    (implies (and (ISA-state-p ISA) (ISA-input-p intr) (addr-p addr))
             (equal (readable-addr? addr (ISA-mem (ISA-step ISA intr)))
                    (readable-addr? addr (ISA-mem ISA))))
  :hints (("Goal" :in-theory (enable ISA-step-functions
                                     ISA-step))))

(encapsulate nil
(local
(defthm readable-addr-MT-final-ISA-help
    (implies (and (ISA-chained-trace-p trace pre)
                  (INST-listp trace)
                  (ISA-state-p pre)
                  (addr-p addr))
             (equal (readable-addr? addr (ISA-mem (trace-final-ISA trace pre)))
                    (readable-addr? addr (ISA-mem pre))))
  :hints (("goal" :in-theory (disable readable-addr-ISA-step))
          (when-found
           (EQUAL (READABLE-ADDR? ADDR (ISA-MEM (INST-POST-ISA (CAR TRACE))))
                  (READABLE-ADDR? ADDR (ISA-MEM (INST-PRE-ISA (CAR TRACE)))))
           (:use (:instance
                  readable-addr-ISA-step (ISA (INST-PRE-ISA (CAR TRACE)))
                  (intr (ISA-input (INST-EXINTR? (CAR TRACE)))))))))))

; Readable-addr-MT-final-ISA shows that the final ISA state has the same
; read protection as the initial ISA state.  Using the previous lemma
; readable-addr-ISA-step, this can be proven by induction.
(defthm readable-addr-MT-final-ISA
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (addr-p addr))
             (equal (readable-addr? addr (ISA-mem (MT-final-ISA MT)))
                    (readable-addr? addr (ISA-mem (MT-init-ISA MT)))))
  :hints (("Goal" :in-theory (enable MT-final-ISA weak-inv
```

```
                                    inv ISA-step-chain-p))))
)

(encapsulate nil
(local
(defthm readable-addr-trace-mem
    (implies (and (INST-listp trace)
                  (addr-p addr)
                  (ISA-state-p ISA)
                  (ISA-chained-trace-p trace ISA))
             (equal (readable-addr? addr
                                    (trace-mem trace (ISA-mem ISA)))
                    (readable-addr? addr (ISA-mem ISA))))
  :hints (("Goal" :in-theory (disable READABLE-ADDR-ISA-STEP))
          (when-found (TRACE-MEM (CDR TRACE)
                                 (ISA-MEM (INST-POST-ISA (CAR TRACE))))
                      (:expand (TRACE-MEM TRACE (ISA-MEM ISA))))
          (when-found
           (EQUAL (READABLE-ADDR? ADDR (ISA-MEM (INST-POST-ISA (CAR TRACE))))
                  (READABLE-ADDR? ADDR (ISA-MEM (INST-PRE-ISA (CAR TRACE)))))
           (:use (:instance
                  readable-addr-ISA-step (ISA (INST-PRE-ISA (CAR TRACE)))
                  (intr (ISA-input (INST-EXINTR? (CAR TRACE))))))))))
(local
(defthm readable-addr-MA-mem-help
    (implies (and (MAETT-p MT)
                  (inv MT MA)
                  (MA-state-p MA)
                  (addr-p addr)
                  (equal (MT-mem MT) (MA-mem MA)))
             (equal (readable-addr? addr (MA-mem MA))
                    (readable-addr? addr (ISA-mem (MT-init-ISA MT)))))
  :hints (("Goal" :in-theory (enable MT-mem
                                     weak-inv inv
                                     ISA-step-chain-p)))
  :rule-classes nil))

; Readable-addr-MA-mem shows that the memory read protection in the
; current MA state is the same as the initial ISA state.
(defthm readable-addr-MA-mem
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (addr-p addr))
             (equal (readable-addr? addr (MA-mem MA))
                    (readable-addr? addr (ISA-mem (MT-init-ISA MT)))))
  :hints (("Goal" :in-theory (enable weak-inv inv
                                     mem-match-p)
                  :use (:instance readable-addr-MA-mem-help))))
) ; encapsulate

(encapsulate nil
(local
(defthm readable-addr-INST-pre-ISA-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (member-equal i trace) (INST-p i)
                  (INST-listp trace)
                  (addr-p addr)
                  (MAETT-p MT) (MA-state-p MA))
     (equal (readable-addr? addr (ISA-mem (INST-pre-ISA i)))
            (readable-addr? addr (ISA-mem (INST-pre-ISA (car trace))))))))
```

```
  :hints ((when-found (INST-PRE-ISA (CAR (CDR TRACE)))
                      (:cases ((consp (cdr trace))))))
  :rule-classes nil))

; Memory read protection does not change during ISA executions.
; Thus, all instructions are executed under the same protection.
(defthm readable-addr-INST-pre-ISA
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (addr-p addr))
             (equal (readable-addr? addr (ISA-mem (INST-pre-ISA i)))
                    (readable-addr? addr (ISA-mem (MT-init-ISA MT)))))
  :hints (("goal" :in-theory (enable INST-in)
                  :use (:instance readable-addr-INST-pre-ISA-help
                                  (trace (MT-trace MT))))
          ("goal'" :cases ((consp (MT-trace MT))))))
)

; Memory write protection does not change after ISA-step.
(defthm writable-addr-ISA-step
    (implies (and (addr-p addr)
                  (ISA-state-p ISA))
             (equal (writable-addr? addr (ISA-mem (ISA-step ISA intr)))
                    (writable-addr? addr (ISA-mem ISA))))
  :hints (("goal" :in-theory (enable ISA-step ISA-step-functions))))

(encapsulate nil
(local
(defthm writable-addr-INST-pre-ISA-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (member-equal i trace) (INST-p i)
                  (INST-listp trace)
                  (addr-p addr)
                  (MAETT-p MT) (MA-state-p MA))
     (equal (writable-addr? addr (ISA-mem (INST-pre-ISA i)))
            (writable-addr? addr (ISA-mem (INST-pre-ISA (car trace))))))
  :hints ((when-found (INST-PRE-ISA (CAR (CDR TRACE)))
                      (:cases ((consp (cdr trace))))))))

; Memory write protection does not change during ISA executions.
; Thus, all instructions are executed under the same protection.
(defthm writable-addr-INST-pre-ISA
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (addr-p addr)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (writable-addr? addr (ISA-mem (INST-pre-ISA i)))
                    (writable-addr? addr (ISA-mem (MT-init-ISA MT)))))
  :hints (("goal" :in-theory (enable INST-in)
                  :use (:instance writable-addr-INST-pre-ISA-help
                                  (trace (MT-trace MT))))
          ("goal'" :cases ((consp (MT-trace MT))))))
)

(encapsulate nil
(local
(defthm writable-addr-MA-mem-help-help
    (implies (and (inv MT MA)
                  (consp trace)
                  (INST-listp trace)
```

```
                (addr-p addr)
                (subtrace-p trace MT)
                (MAETT-p MT) (MA-state-p MA))
     (equal (writable-addr? addr
                            (trace-mem trace
                                       (ISA-mem (INST-pre-ISA (car trace)))))
            (writable-addr? addr (ISA-mem (INST-pre-ISA (car trace))))))
  :hints ((when-found (INST-PRE-ISA (CAR (CDR TRACE)))
                      (:cases ((consp (cdr trace))))))
  :rule-classes nil))

(local
(defthm writable-addr-MA-mem-help
    (implies (and (inv MT MA)
                  (addr-p addr)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (writable-addr? addr (MT-mem MT))
                    (writable-addr? addr (ISA-mem (MT-init-ISA MT)))))
  :hints (("goal" :in-theory (e/d (MT-mem)
                                  (MT-mem-=-MA-mem))
                  :use (:instance writable-addr-MA-mem-help-help
                                  (trace (MT-trace MT)))))
  :rule-classes nil))

; writable-addr-MA-mem shows that the memory write protection in the
; current MA state is the same as in the initial ISA state.
(defthm writable-addr-MA-mem
    (implies (and (inv MT MA)
                  (addr-p addr)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (writable-addr? addr (MA-mem MA))
                    (writable-addr? addr (ISA-mem (MT-init-ISA MT)))))
  :hints (("goal" :in-theory (enable MT-mem-=-MA-mem)
                  :use (:instance writable-addr-MA-mem-help))))
)

;;  Lemmas about supervisor/user mode.
(encapsulate nil
(local
(defthm MT-SRF-=-ISA-before-MT-non-commit-trace-help
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (INST-listp trace)
                  (subtrace-p trace MT)
                  (tail-p (MT-non-commit-trace MT) trace))
             (equal (trace-SRF trace (ISA-SRF pre))
                    (ISA-SRF (ISA-at-tail (MT-non-commit-trace MT)
                                          trace pre))))))

;; The ISA state at the commit boundary and the current
;; MA state have the same special register file
(defthm MT-SRF-=-ISA-before-MT-non-commit-trace
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA))
             (equal (ISA-SRF (ISA-before (MT-non-commit-trace MT) MT))
                    (MT-SRF MT)))
  :hints (("Goal" :in-theory (e/d (ISA-before MT-SRF)
                                  (MT-SRF-=-MA-SRF)))))

)

;; Unless an exception or an context synchronization takes place,
```

```
;; supervisor/user mode does not change.
(defthm sreg-su-INST-pre-ISA-of-non-exintr-context-sync
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (INST-p i)
                  (not (b1p (INST-EXCPT? i)))
                  (not (b1p (INST-context-sync? i)))
                  (not (b1p (ISA-input-exint intr))))
             (equal (SRF-su (ISA-SRF (ISA-step (INST-pre-ISA i) intr)))
                    (SRF-su (ISA-SRF (INST-pre-ISA i)))))
  :hints (("Goal" :in-theory (enable ISA-step-functions ISA-step
                                     lift-b-ops
                                     INST-function-def
                                     decode-illegal-inst?
                                     decode logbit* rdb
                                     supervisor-mode?
                                     MT-def-non-rec-functions
                                     equal-b1p-converter
                                     write-sreg
                                     ))))

(encapsulate nil
(local
(defthm local-help-lemma2
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (INST-p i)
                  (not (commit-stg-p (INST-stg i)))
                  (not (retire-stg-p (INST-stg i)))
                  (not (b1p (INST-start-specultv? i)))
                  (not (b1p (ISA-input-exint intr))))
             (equal (SRF-su (ISA-SRF
                             (ISA-step (INST-pre-ISA i) intr)))
                    (SRF-su (ISA-SRF (INST-pre-ISA i)))))
  :hints (("goal" :in-theory (enable INST-start-specultv? lift-b-ops)))))

(local
(defthm local-help-lemma1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-listp trace)
                  (consp trace)
                  (subtrace-p trace MT)
                  (no-commit-INST-p trace)
                  (not (b1p (trace-specultv? trace))))
     (equal (SRF-su (ISA-SRF (trace-final-ISA trace
                                              (INST-pre-ISA (car trace)))))
            (SRF-su (ISA-SRF (INST-pre-ISA (car trace))))))
  :hints (("Goal" :in-theory (enable lift-b-ops
                                     dispatched-p committed-p
                                     not-inst-specultv-INST-in-if-committed
                                     INST-exintr-INST-in-if-not-retired))
          (when-found (INST-PRE-ISA (car (cdr TRACE)))
                      (:cases ((consp (cdr trace)))))))
  :rule-classes nil))

; The supervisor/user mode is the same at the final ISA state and the
; pre-ISA state of the first uncommitted instruction.
```

```
(defthm SRF-su-INST-pre-ISA-MT-non-commit-trace
    (implies (and (inv MT MA)
                  (consp (MT-non-commit-trace MT))
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (not (b1p (MT-specultv? MT))))
     (equal (SRF-su (ISA-SRF (INST-pre-ISA (car (MT-non-commit-trace MT)))))
            (SRF-su (ISA-SRF (MT-final-ISA MT)))))
  :hints (("Goal"
           :use ((:instance local-help-lemma1
                            (trace (MT-non-commit-trace MT)))
                 (:instance trace-final-ISA-subtrace
                            (trace (MT-non-commit-trace MT))
                            (pre (INST-pre-ISA
                                  (car (MT-non-commit-trace MT)))))))))
) ;encapsulate

;; Supervisor/user mode in the pre-ISA state of the first uncommitted
;; instruction, and the final ISA state of MT are the same, provided
;; that no instructions currently being executed cause internal
;; exceptions.  MT-specultv? checks if currently executed instructions
;; cause exceptions or misprediction of branches.
(defthm SRF-su-MT-final-ISA-=-MT-non-commit-trace
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (MT-specultv? MT))))
     (equal (SRF-su (ISA-SRF (MT-final-ISA MT)))
            (SRF-su (ISA-SRF (ISA-before (MT-non-commit-trace MT) MT)))))
  :hints (("Goal" :cases ((consp (MT-non-commit-trace MT))))))

(encapsulate nil
(local
(defthm SRF-su-MT-final-ISA-help
    (implies (and (MA-state-p MA)
                  (inv MT MA)
                  (MAETT-p MT)
                  (not (b1p (MT-specultv? MT))))
             (equal (SRF-su (ISA-SRF (MT-final-ISA MT)))
                    (SRF-su (MT-SRF MT)))))
)

;; Given that no instructions currently being executed cause
;; exceptions, the supervisor/user mode in the current MA and the
;; final ISA state of MT are the same.
(defthm SRF-su-ISA-SRF-MT-final-ISA
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (not (b1p (MT-specultv? MT))))
             (equal (SRF-su (ISA-SRF (MT-final-ISA MT)))
                    (SRF-su (MA-SRF MA))))
  :hints (("goal" :in-theory (enable weak-inv inv
                                     SRF-match-p))))
)

;; Lemmas about side effects of memory operations.

;; Provided that the processor is not executing instructions
;; speculatively and has not detected any exceptions, an
;; instruction fetch error that occurs in the final ISA state of MT also
;; occurs in the current MA state, when it tries to fetch the next
;; instruction.
```

```
(defthm read-error-MT-final-ISA
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (addr-p addr)
                  (not (b1p (MT-specultv? MT))))
             (equal (read-error? addr
                                 (ISA-mem (MT-final-ISA MT))
                                 (SRF-su (MA-SRF MA)))
                    (read-error? addr (MA-mem MA) (SRF-su (MA-SRF MA)))))
  :hints (("goal" :in-theory (enable read-error?))))

;; Only an store instruction, without an exception and an external
;; interrupt, changes the memory.  Otherwise, the memory value remains
;; the same.
(defthm read-mem-ISA-step-if-not-ISA-store-inst-p
    (implies (and (ISA-state-p ISA)
                  (addr-p addr)
                  (not (and (ISA-store-inst-p ISA)
                            (equal (ISA-store-addr ISA) addr)
                            (not (ISA-excpt-p ISA))
                            (not (b1p (ISA-input-exint intr))))))
             (equal (read-mem addr (ISA-mem (ISA-step ISA intr)))
                    (read-mem addr (ISA-mem ISA))))
  :hints (("goal" :in-theory (enable ISA-store-addr ISA-store-inst-p
                                     ISA-functions
                                     store-inst-p
                                     ISA-step
                                     ISA-step-functions))))

(encapsulate nil
(local
(defthm read-mem-MA-mem-if-MT-no-write-at-induction
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (consp trace)
                  (INST-listp trace)
                  (addr-p addr)
                  (trace-no-write-at addr trace))
             (equal (read-mem addr
                              (trace-mem trace
                                         (ISA-mem (INST-pre-ISA (car trace)))))
                    (read-mem addr (ISA-mem (INST-pre-ISA (car trace))))))
  :hints ((when-found (ISA-MEM (INST-PRE-ISA (car (CDR TRACE))))
                      (:cases ((consp (cdr trace))))))
  :rule-classes nil))

(local
(defthm read-mem-MA-mem-if-MT-no-write-at-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (addr-p addr)
                  (MT-no-write-at addr MT))
             (equal (read-mem addr (MT-mem MT))
                    (read-mem addr (ISA-mem (MT-init-ISA MT)))))
  :hints (("Goal" :in-theory (e/d (MT-mem MT-no-write-at)
                                  (MT-MEM-=-MA-MEM))
                  :use (:instance read-mem-MA-mem-if-MT-no-write-at-induction
                                  (trace (MT-trace MT))))
          ("Goal'" :cases ((consp (MT-trace MT)))))
  :rule-classes nil)
```

```
)

;; Provided that no instructions in MT modifies the memory value at address
;; addr,  the memory value remains the same from the initial state.
;; This lemma is useful in proving that the instruction fetched by the MA
;; is the same as in the ISA, provided that no self-modification of the
;; program occurs.
(defthm read-mem-MA-mem-if-MT-no-write-at
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (addr-p addr)
                  (MT-no-write-at addr MT))
             (equal (read-mem addr (MA-mem MA))
                    (read-mem addr (ISA-mem (MT-init-ISA MT)))))
  :hints (("goal" :use (:instance read-mem-MA-mem-if-MT-no-write-at-help))))
) ;encapsulate

(encapsulate nil
(local
(defthm read-mem-MT-final-ISA-if-MT-no-write-at-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (consp trace)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (addr-p addr)
                  (MAETT-p MT) (MA-state-p MA)
                  (trace-no-write-at addr trace))
             (equal (read-mem addr
                              (ISA-mem (trace-final-ISA
                                        trace
                                        (INST-pre-ISA (car trace)))))
                    (read-mem addr (ISA-mem (INST-pre-ISA (car trace))))))
  :hints (("Goal" :induct t)
          (when-found (INST-PRE-ISA (CAR (CDR TRACE)))
                      (:cases ((consp (cdr trace))))))
  :rule-classes nil))

;; Provided that there is no memory write operation on address addr, the
;; memory value at addr at the initial and final ISA states are the same.
(defthm read-mem-MT-final-ISA-if-MT-no-write-at
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (addr-p addr)
                  (MT-no-write-at addr MT))
             (equal (read-mem addr (ISA-mem (MT-final-ISA MT)))
                    (read-mem addr (ISA-mem (MT-init-ISA MT)))))
  :hints (("Goal" :in-theory (enable MT-final-ISA
                                     MT-no-write-at)
                  :use (:instance read-mem-MT-final-ISA-if-MT-no-write-at-help
                                  (trace (MT-trace MT))))))
)
```

D.6.2 modifier.lisp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MI-inv.lisp
; Author  Jun Sawada, University of Texas at Austin
;
;  This book proves theorems about register modifiers and memory
```

```
;  modifiers.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(in-package "ACL2")

(include-book "MA2-lemmas")
(include-book "MAETT-lemmas")
(include-book "memory-inv")

(deflabel begin-modifier)
; This book proves basic lemmas about register modifiers and memory
; modifiers.
;  Index
;    Lemmas about register modifiers
;      Basic Lemmas
;      Relation between register modifiers and register reference table
;      Behavior of register modifier in the ISA model
;      Other more complex lemmas.
;    Lemmas about memory modifiers
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Lemmas about register modifiers
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Start of basic lemmas about register modifiers.

; A register modifier in the current state is a register modifier in
; the next state.
(defthm reg-modifier-step-inst
    (equal (reg-modifier-p idx (step-inst i MT MA sigs))
           (reg-modifier-p idx i))
  :hints (("Goal" :in-theory (enable reg-modifier-p))))

; A special register modifier in the current state is a register
; modifier in the next state.
(defthm sreg-modifier-step-inst
    (equal (sreg-modifier-p idx (step-inst i MT MA sigs))
           (sreg-modifier-p idx i))
  :hints (("Goal" :in-theory (enable sreg-modifier-p))))

; Some trivial lemmas.

; relation between the register modifiers and uncommitted register modifiers.
(defthm trace-exist-uncommitted-LRM-if-exist-reg-modifier
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (trace-exist-LRM-before-p i r trace)
                  (not (committed-p (trace-LRM-before i r trace))))
             (trace-exist-uncommitted-LRM-before-p i r trace)))

; If there are register modifiers, and the last register modifier is
; not committed, then there exists uncommitted register modifiers.
(defthm exist-uncommitted-LRM-if-exist-reg-modifier
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (exist-LRM-before-p i r MT)
                  (not (committed-p (LRM-before i r MT))))
             (exist-uncommitted-LRM-before-p i r MT))
  :hints (("goal" :in-theory (enable exist-uncommitted-LRM-before-p
                                     exist-LRM-before-p
                                     LRM-before))))

(defthm trace-exist-uncommitted-LSRM-if-exist-sreg-modifier
```

```
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (trace-exist-LSRM-before-p i r trace)
                  (not (committed-p (trace-LSRM-before i r trace))))
             (trace-exist-uncommitted-LSRM-before-p i r trace)))

; If there are special register modifiers, and the last special
; register modifier is not committed, then there exists uncommitted
; special register modifiers.
(defthm exist-uncommitted-LSRM-if-exist-reg-modifier
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (exist-LSRM-before-p i r MT)
                  (not (committed-p (LSRM-before i r MT))))
             (exist-uncommitted-LSRM-before-p i r MT))
  :hints (("goal" :in-theory (enable exist-uncommitted-LSRM-before-p
                                     exist-LSRM-before-p
                                     LSRM-before))))

(in-theory (disable trace-exist-uncommitted-LRM-if-exist-reg-modifier
                    trace-exist-uncommitted-LSRM-if-exist-sreg-modifier))

(defthm exist-uncommitted-LRM-if-commit-car
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (not (committed-p (car trace)))
                  (trace-exist-LRM-before-p i r (cdr trace)))
             (trace-exist-uncommitted-LRM-before-p i r (cdr trace)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
   (implies (and (inv MT MA)
                 (MAETT-p MT) (MA-state-p MA)
                 (subtrace-p trace MT)
                 (INST-listp trace)
                 (not (committed-p (car trace)))
                 (not (trace-exist-uncommitted-LRM-before-p i r
                                                            (cdr trace))))
            (not (trace-exist-LRM-before-p i r (cdr trace)))))))

(defthm exist-uncommitted-LSRM-if-commit-car
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (not (committed-p (car trace)))
                  (trace-exist-LSRM-before-p i r (cdr trace)))
             (trace-exist-uncommitted-LSRM-before-p i r (cdr trace)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (not (committed-p (car trace)))
                  (not (trace-exist-uncommitted-LSRM-before-p i r
                                                              (cdr trace))))
```

```
            (not (trace-exist-LSRM-before-p i r (cdr trace)))))))

(encapsulate nil
(local
(defthm inst-in-LRM-before-help
    (implies (trace-exist-LRM-before-p i rname trace)
             (member-equal (trace-LRM-before i rname trace)
                           trace))))

(defthm inst-in-LRM-before
    (implies (exist-LRM-before-p i rname MT)
             (INST-in (LRM-before i rname MT) MT))
  :hints (("goal" :in-theory (enable INST-in LRM-before
                                     exist-LRM-before-p))))
)

(encapsulate  nil
(local
(defthm inst-in-LSRM-before-help
    (implies (trace-exist-LSRM-before-p i rname trace)
             (member-equal (trace-LSRM-before i rname trace)
                           trace))))

(defthm inst-in-LSRM-before
    (implies (exist-LSRM-before-p i rname MT)
             (INST-in (LSRM-before i rname MT) MT))
  :hints (("goal" :in-theory (enable INST-in LSRM-before
                                     exist-LSRM-before-p))))
)

; Register modifiers are INSTs in MT.
(encapsulate nil
(local
(defthm inst-in-LRM-in-ROB-help
    (implies (trace-exist-LRM-in-ROB-p rname trace)
             (member-equal (trace-LRM-in-ROB rname trace)
                           trace))))

(defthm inst-in-LRM-in-ROB
    (implies (exist-LRM-in-ROB-p rname MT)
             (INST-in (LRM-in-ROB rname MT) MT))
  :hints (("goal" :in-theory (enable INST-in exist-LRM-in-ROB-p
                                     LRM-in-ROB))))

)

(encapsulate nil
(local
(defthm inst-in-LSRM-in-ROB-help
    (implies (trace-exist-LSRM-in-ROB-p rname trace)
             (member-equal (trace-LSRM-in-ROB rname trace)
                           trace))))

(defthm inst-in-LSRM-in-ROB
    (implies (exist-LSRM-in-ROB-p rname MT)
             (INST-in (LSRM-in-ROB rname MT) MT))
  :hints (("goal" :in-theory (enable INST-in exist-LSRM-in-ROB-p
                                     LSRM-in-ROB))))
)

(encapsulate nil
(local
```

```
 (defthm LRM-before-differs-from-i-help
    (implies (trace-exist-LRM-before-p i rname MT)
             (not (equal (trace-LRM-before i rname MT) i)))))

; The last register modifier before i is not i.
(defthm LRM-before-differs-from-i
    (implies (exist-LRM-before-p i rname MT)
             (not (equal (LRM-before i rname MT) i)))
  :hints (("goal" :in-theory (enable exist-LRM-before-p
                                     LRM-before))))
)

(encapsulate nil
(local
 (defthm LSRM-before-differs-from-i-help
    (implies (trace-exist-LSRM-before-p i rname MT)
             (not (equal (trace-LSRM-before i rname MT) i)))))

; The last special register modifier before i is not i.
(defthm LSRM-before-differs-from-i
    (implies (exist-LSRM-before-p i rname MT)
             (not (equal (LSRM-before i rname MT) i)))
  :hints (("goal" :in-theory (enable exist-LSRM-before-p
                                     LSRM-before))))
)

(encapsulate nil
(local
 (defthm INST-in-order-LRM-before-help
    (implies (and (member-equal i trace)
                  (trace-exist-LRM-before-p I rname trace))
             (member-in-order (trace-LRM-before i rname trace)
                              i trace))
   :hints (("goal" :in-theory (enable member-in-order*)))))

; The last register modifier before i precedes i in program order.
(defthm INST-in-order-LRM-before
    (implies (and (INST-in i MT) (exist-LRM-before-p I rname MT))
             (INST-in-order-p (LRM-before i rname MT) i MT))
  :hints (("goal" :in-theory (enable LRM-before INST-in
                                     INST-in-order-p exist-LRM-before-p))))
)

(encapsulate nil
(local
 (defthm INST-in-order-LSRM-before-help
    (implies (and (member-equal i trace)
                  (trace-exist-LSRM-before-p I rname trace))
             (member-in-order (trace-LSRM-before i rname trace)
                              i trace))
   :hints (("goal" :in-theory (enable member-in-order*)))))

; The last special register modifier before i precedes i in program order.
(defthm INST-in-order-LSRM-before
    (implies (and (INST-in i MT) (exist-LSRM-before-p I rname MT))
             (INST-in-order-p (LSRM-before i rname MT) i MT))
  :hints (("goal" :in-theory (enable exist-LSRM-before-p INST-in
                                     INST-in-order-p LSRM-before))))
)

(encapsulate nil
(local
```

```
(defthm LRM-is-last-help-help
    (implies (trace-exist-LRM-before-p i r trace)
             (member-equal (trace-LRM-before i r trace)
                           trace))))

(local
(defthm LRM-is-last-help
    (implies (and (distinct-member-p trace)
                  (member-equal j trace)
                  (reg-modifier-p r j)
                  (member-in-order j i trace)
                  (not (equal (trace-LRM-before i r trace) j)))
             (member-in-order j (trace-LRM-before i r trace)
                              trace))
  :hints (("goal" :in-theory (enable member-in-order*)))))

; If j is a r-register modifier, and j precedes i in program
; order, then j is the last r-register modifier, or the last memory
; modifier exists between j and i.
(defthm LRM-is-last
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (reg-modifier-p r j)
                  (INST-in-order-p j i MT)
                  (not (equal (LRM-before i r MT) j)))
             (INST-in-order-p j (LRM-before i r MT) MT))
  :hints (("goal" :in-theory (enable INST-in-order-p LRM-before
                                     MT-distinct-inst-p
                                     inv weak-inv
                                     INST-in))))
)

(encapsulate nil
(local
(defthm LSRM-is-last-help-help
    (implies (trace-exist-LSRM-before-p i sr trace)
             (member-equal (trace-LSRM-before i sr trace)
                           trace))))

(local
(defthm LSRM-is-last-help
    (implies (and (distinct-member-p trace)
                  (member-equal j trace)
                  (sreg-modifier-p sr j)
                  (member-in-order j i trace)
                  (not (equal (trace-LSRM-before i sr trace) j)))
             (member-in-order j (trace-LSRM-before i sr trace)
                              trace))
  :hints (("goal" :in-theory (enable member-in-order*)))))

; If j is a r-register modifier, and j precedes i in program
; order, then j is the last r-register modifier, or the last memory
; modifier exists between j and i.
(defthm LSRM-is-last
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (sreg-modifier-p sr j)
                  (INST-in-order-p j i MT)
```

```
                (not (equal (LSRM-before i sr MT) j)))
           (INST-in-order-p j (LSRM-before i sr MT) MT))
  :hints (("goal" :in-theory (enable INST-in-order-p LSRM-before
                                     MT-distinct-inst-p
                                     inv weak-inv
                                     INST-in))))
)

(encapsulate nil
(local
(defthm reg-modifier-p-LRM-before-help
    (implies (trace-exist-LRM-before-p I rname trace)
             (reg-modifier-p rname
                             (trace-LRM-before I rname trace)))))

; The last register modifier is a register modifier.
(defthm reg-modifier-p-LRM-before
    (implies (exist-LRM-before-p I rname MT)
             (reg-modifier-p rname (LRM-before I rname MT)))
  :hints (("goal" :in-theory (enable exist-LRM-before-p
                                     LRM-before))))
)

(encapsulate nil
(local
(defthm sreg-modifier-p-LSRM-before-help
    (implies (trace-exist-LSRM-before-p I rname trace)
             (sreg-modifier-p rname
                              (trace-LSRM-before I rname trace)))))

; The last special register modifier is a special register modifier.
(defthm sreg-modifier-p-LSRM-before
    (implies (exist-LSRM-before-p I rname MT)
             (sreg-modifier-p rname (LSRM-before I rname MT)))
  :hints (("goal" :in-theory (enable exist-LSRM-before-p
                                     LSRM-before))))
)

(encapsulate nil
(local
(defthm reg-modifier-p-LRM-in-ROB-help
    (implies (trace-exist-LRM-in-ROB-p idx trace)
             (reg-modifier-p idx (trace-LRM-in-ROB idx trace)))))

; The last register modifier in the ROB is a register modifier.
(defthm reg-modifier-p-LRM-in-ROB
    (implies (exist-LRM-in-ROB-p idx MT)
             (reg-modifier-p idx (LRM-in-ROB idx MT)))
  :hints (("Goal" :in-theory (enable exist-LRM-in-ROB-p
                                     LRM-in-ROB))))
)

(encapsulate nil
(local
(defthm sreg-modifier-p-LSRM-in-ROB-help
    (implies (trace-exist-LSRM-in-ROB-p idx trace)
             (sreg-modifier-p idx
                              (trace-LSRM-in-ROB idx trace)))))

; The last special register modifier in the ROB is a special register
; modifier.
(defthm sreg-modifier-p-LSRM-in-ROB
```

```
    (implies (exist-LSRM-in-ROB-p idx MT)
             (sreg-modifier-p idx (LSRM-in-ROB idx MT)))
  :hints (("Goal" :in-theory (enable exist-LSRM-in-ROB-p
                                     LSRM-in-ROB))))
)

; A register modifier is a write-back instruction.
(defthm INST-writeback-p-reg-modifier
    (implies (reg-modifier-p rname i)
             (INST-writeback-p i))
  :hints (("goal" :in-theory (enable reg-modifier-p
                                     INST-function-def
                                     decode logbit* rdb lift-b-ops))))

; A special register modifier is a write-back instruction.
(defthm INST-writeback-p-sreg-modifier
    (implies (sreg-modifier-p rname i)
             (INST-writeback-p i))
  :hints (("goal" :in-theory (enable sreg-modifier-p
                                     INST-function-def lift-b-ops
                                     decode logbit* rdb))))
(in-theory (disable INST-writeback-p-reg-modifier
                    INST-writeback-p-sreg-modifier))

; The last register modifier is a write-back instruction.
(defthm INST-writeback-p-LRM-before
    (implies (and (MAETT-p MT) (INST-p i)
                  (exist-LRM-before-p I rname MT))
             (INST-writeback-p (LRM-before I rname MT)))
  :hints (("goal" :in-theory (disable INST-writeback-p-reg-modifier)
                  :use (:instance INST-writeback-p-reg-modifier
                                  (i (LRM-BEFORE I RNAME MT))))))

; The last special register modifier is a write-back instruction.
(defthm INST-writeback-p-LSRM-before
    (implies (exist-LSRM-before-p I rname MT)
             (INST-writeback-p (LSRM-before I rname MT)))
  :hints (("goal" :in-theory (disable INST-writeback-p-sreg-modifier)
                  :use (:instance INST-writeback-p-sreg-modifier
                                  (i (LSRM-BEFORE I RNAME MT))))))

; The destination register of the last register modifier of
; register idx is idx itself.
(defthm INST-dest-reg-LRM-in-ROB
    (implies (exist-LRM-in-ROB-p idx MT)
             (equal (INST-dest-reg (LRM-in-ROB idx MT)) idx))
  :hints (("Goal" :use (:instance REG-MODIFIER-P-LRM-IN-ROB)
                  :in-theory (e/d (reg-modifier-p)
                                  (REG-MODIFIER-P-LRM-IN-ROB)))))

; Similar to INST-dest-reg-LRM-in-ROB.
(defthm INST-dest-sreg-LRM-in-ROB
    (implies (exist-LSRM-in-ROB-p idx MT)
             (equal (INST-dest-reg (LSRM-in-ROB idx MT)) idx))
  :hints (("Goal" :use (:instance SREG-MODIFIER-P-LSRM-IN-ROB)
                  :in-theory (e/d (sreg-modifier-p)
                                  (SREG-MODIFIER-P-LSRM-IN-ROB)))))

;; End of basic lemmas about register modifiers.

;; Start of the lemmas about the relation between the register modifiers
;; and the register reference table.
```

```
(encapsulate nil
(local
(defthm consistent-reg-ref-if-consistent-reg-tbl-under
    (implies (and (consistent-reg-tbl-under ub MT MA)
                  (integerp ub) (integerp rix)
                  (<= 0 rix) (< rix ub))
             (consistent-reg-ref-p rix MT MA))))

(local
  (defthm not-reg-modifier-in-rob-if-not-reg-ref-wait-help
      (implies (and (consistent-reg-ref-p rname MT MA)
                    (not (b1p (MT-specultv-at-dispatch? MT)))
                    (not (b1p (MT-modified-at-dispatch? MT))))
               (iff (exist-LRM-in-ROB-p rname MT)
                    (b1p (reg-ref-wait?
                          (reg-tbl-nth rname (DQ-reg-tbl (MA-DQ MA)))))))
    :hints (("goal" :in-theory (enable consistent-reg-ref-p)))))

; Wait? field of the corresponding entry in the register
; reference table is 1 iff register modifiers of register rname
; are currently stored in the ROB.
(defthm not-reg-modifier-in-rob-if-not-reg-ref-wait
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (rname-p rname)
                  (not (b1p (MT-specultv-at-dispatch? MT)))
                  (not (b1p (MT-modified-at-dispatch? MT))))
             (iff (exist-LRM-in-ROB-p rname MT)
                  (b1p (reg-ref-wait?
                        (reg-tbl-nth rname (DQ-reg-tbl (MA-DQ MA)))))))
  :hints (("goal" :in-theory (enable inv consistent-reg-tbl-p)
                  :restrict ((not-reg-modifier-in-rob-if-not-reg-ref-wait-help
                              ((MA MA)))))))
(local
(defthm reg-ref-tag-points-to-LRM-in-ROB-help
    (implies (and (consistent-reg-ref-p rname MT MA)
                  (not (b1p (MT-specultv-at-dispatch? MT)))
                  (not (b1p (MT-modified-at-dispatch? MT)))
                  (b1p (reg-ref-wait? (reg-tbl-nth rname
                                                   (DQ-reg-tbl (MA-DQ MA))))))
             (equal (reg-ref-tag (reg-tbl-nth rname (DQ-reg-tbl (MA-DQ MA))))
                    (INST-tag (LRM-in-ROB rname MT))))
  :hints (("goal" :in-theory (enable consistent-reg-ref-p lift-b-ops)))))

; If wait? field is 1 in the register reference table,
; field tag of a register reference table entry designates the
; ROB entry where the last register modifier is stored.
(defthm reg-ref-tag-points-to-LRM-in-ROB
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (rname-p rname)
                  (not (b1p (MT-specultv-at-dispatch? MT)))
                  (not (b1p (MT-modified-at-dispatch? MT)))
                  (b1p (reg-ref-wait? (reg-tbl-nth rname
                                                   (DQ-reg-tbl (MA-DQ MA))))))
             (equal (reg-ref-tag (reg-tbl-nth rname (DQ-reg-tbl (MA-DQ MA))))
                    (INST-tag (LRM-in-ROB rname MT))))
  :hints (("goal" :in-theory (enable inv consistent-reg-tbl-p)
                  :restrict ((reg-ref-tag-points-to-LRM-in-ROB-help
                              ((MT MT)))))))
)
```

```
(encapsulate nil
(local
  (defthm not-sreg-modifier-in-rob-if-not-sreg-ref-wait-help
      (implies (and (consistent-sreg-ref-p rname MT MA)
                    (not (b1p (MT-specultv-at-dispatch? MT)))
                    (not (b1p (MT-modified-at-dispatch? MT))))
               (iff (exist-LSRM-in-ROB-p rname MT)
                    (b1p (reg-ref-wait?
                          (sreg-tbl-nth rname (DQ-sreg-tbl (MA-DQ MA)))))))
    :hints (("goal" :in-theory (enable consistent-sreg-ref-p)))))

; Register modifiers of special register rname are in the ROB if
; the wait field of the corresponding entry in the register reference
; table is 1.
(defthm not-sreg-modifier-in-rob-if-not-sreg-ref-wait
    (implies (and (inv MT MA)
                  (srname-p rname)
                  (not (b1p (MT-specultv-at-dispatch? MT)))
                  (not (b1p (MT-modified-at-dispatch? MT))))
             (iff (exist-LSRM-in-ROB-p rname MT)
                  (b1p (reg-ref-wait?
                        (sreg-tbl-nth rname (DQ-sreg-tbl (MA-DQ MA)))))))
  :hints (("goal" :in-theory (enable inv consistent-sreg-tbl-p
                                     srname-p)
                  :cases ((equal rname 0) (equal rname 1)))))
)

(encapsulate nil
(local
(defthm sreg-ref-tag-points-to-LSRM-in-ROB-help
    (implies (and (consistent-sreg-ref-p rname MT MA)
                  (not (b1p (MT-specultv-at-dispatch? MT)))
                  (not (b1p (MT-modified-at-dispatch? MT)))
                  (b1p (reg-ref-wait?
                        (sreg-tbl-nth rname (DQ-sreg-tbl (MA-DQ MA))))))
     (equal (reg-ref-tag (sreg-tbl-nth rname (DQ-sreg-tbl (MA-DQ MA))))
            (INST-tag (LSRM-in-ROB rname MT))))
  :hints (("goal" :in-theory (enable consistent-sreg-ref-p lift-b-ops)))))

; Field tag of the special register reference table entry designates
; the ROB entry to which the last modifier of the special register is
; assigned.
(defthm reg-ref-tag-points-to-LSRM-in-ROB
    (implies (and (inv MT MA)
                  (b1p (reg-ref-wait?
                        (sreg-tbl-nth rname (DQ-sreg-tbl (MA-DQ MA)))))
                  (srname-p rname)
                  (not (b1p (MT-specultv-at-dispatch? MT)))
                  (not (b1p (MT-modified-at-dispatch? MT))))
             (equal (reg-ref-tag
                     (sreg-tbl-nth rname (DQ-sreg-tbl (MA-DQ MA))))
                    (INST-tag (LSRM-in-ROB rname MT))))
  :hints (("goal" :in-theory (enable inv consistent-sreg-tbl-p
                                     srname-p)
                  :cases ((equal rname 0) (equal rname 1)))))
)
;; End of the lemmas about the relation between the register modifiers
;; and the register reference table.

;; Lemmas about the behavior of register modifiers in the ISA model.

; If instruction i is not a write-back instruction, it does not modify
```

```
; the register file in ISA executions.
;
; Note: the same lemma is proven in ISA-comp.lisp.  We want to move
; it to one of the books containing shared lemmas, but we have not
; yet found the right place.
(defthm ISA-RF-ISA-step-if-not-INST-wb
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (not (b1p (INST-wb? i))))
             (equal (ISA-RF (ISA-step (INST-pre-ISA i) intr))
                    (ISA-RF (INST-pre-ISA i))))
  :hints (("goal" :in-theory (enable ISA-step-functions ISA-step
                                     INST-wb? INST-cntlv
                                     INST-opcode
                                     opcode-inst-type))))

; If (INST-wb-sreg? i) is 1, instruction i does not modify
; general register file in the ISA execution. WB-sreg? specifies whether
; the instruction modifies a general register or a special register.
; Wb-SRF? is 1 iff the instruction is modifying a special register.
(defthm read-reg-ISA-step-if-INST-wb-sreg
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (rname-p rname)
                  (b1p (INST-wb-sreg? i)))
     (equal (read-reg rname (ISA-RF (ISA-step (INST-pre-ISA i) intr)))
            (read-reg rname (ISA-RF (INST-pre-ISA i)))))
  :hints (("goal" :in-theory (enable INST-wb-sreg? decode INST-cntlv
                                     lift-b-ops rdb logbit*
                                     INST-opcode
                                     ISA-step ISA-step-functions))))

; If (INST-dest-reg i) is different from rname, instruction i does
; not modify register rname.
(defthm read-reg-ISA-step-if-not-INST-dest-reg
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (rname-p rname)
                  (not (equal (INST-dest-reg I) rname)))
     (equal (read-reg rname (ISA-RF (ISA-step (INST-pre-ISA i) intr)))
            (read-reg rname (ISA-RF (INST-pre-ISA i)))))
  :hints (("goal" :in-theory (enable INST-DEST-REG
                                     INST-rc INST-ra
                                     ISA-step ISA-step-functions))))

; If instruction i is not a register modifier of register rname,
; i does not modify the value of register rname.
(defthm read-reg-ISA-step-non-reg-modifier
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (rname-p rname)
                  (not (reg-modifier-p rname i)))
     (equal (read-reg rname (ISA-RF (ISA-step (INST-pre-ISA i) intr)))
            (read-reg rname (ISA-RF (INST-pre-ISA i)))))
  :hints (("goal" :in-theory (enable  reg-modifier-p))))

; If (INST-wb? i) is 0, instruction i does not modify
; special register file.
```

```
(defthm read-sreg-ISA-step-if-not-INST-wb
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i)  (INST-in i MT)
                  (rname-p rname)
                  (not (b1p (INST-wb? i)))
                  (not (b1p (INST-excpt? i)))
                  (not (b1p (INST-context-sync? i)))
                  (not (b1p (ISA-input-exint intr))))
     (equal (read-sreg rname (ISA-SRF (ISA-step (INST-pre-ISA i) intr)))
            (read-sreg rname (ISA-SRF (INST-pre-ISA i)))))
  :hints (("goal" :in-theory (enable decode
                                     lift-b-ops rdb logbit*
                                     equal-b1p-converter
                                     DECODE-ILLEGAL-INST?
                                     INST-function-def
                                     ISA-step ISA-step-functions))))

; If (INST-wb-sreg? i) is 0, instruction i does not modify
; the special register file.
(defthm read-sreg-ISA-step-if-not-INST-wb-sreg
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (rname-p rname)
                  (not (b1p (INST-wb-sreg? i)))
                  (not (b1p (INST-excpt? i)))
                  (not (b1p (INST-context-sync? i)))
                  (not (b1p (ISA-input-exint intr))))
     (equal (read-sreg rname (ISA-SRF (ISA-step (INST-pre-ISA i) intr)))
            (read-sreg rname (ISA-SRF (INST-pre-ISA i)))))
  :hints (("goal" :in-theory (enable decode lift-b-ops rdb logbit*
                                     DECODE-ILLEGAL-INST?
                                     INST-function-def
                                     ISA-step ISA-step-functions))))

; If (INST-dest-reg i) is different from rname, instruction i does
; not modify special register rname.
(defthm read-sreg-ISA-step-if-not-INST-dest-reg
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (srname-p rname)
                  (not (equal (INST-dest-reg I) rname))
                  (not (b1p (INST-excpt? i)))
                  (not (b1p (INST-context-sync? i)))
                  (not (b1p (ISA-input-exint intr))))
     (equal (read-sreg rname (ISA-SRF (ISA-step (INST-pre-ISA i) intr)))
            (read-sreg rname (ISA-SRF (INST-pre-ISA i)))))
  :hints (("goal" :in-theory (enable INST-DEST-REG
                                     decode rdb logbit* lift-b-ops
                                     equal-b1p-converter
                                     INST-function-def DECODE-ILLEGAL-INST?
                                     INST-rc INST-ra
                                     ISA-step ISA-step-functions))))

; If i is not a register modifier of a special register rname, i does not
; modify the special register.
(defthm read-sreg-ISA-step-non-sreg-modifier
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
```

```
                (srname-p rname)
                (not (sreg-modifier-p rname i))
                (not (b1p (INST-excpt? i)))
                (not (b1p (INST-context-sync? i)))
                (not (b1p (ISA-input-exint intr))))
     (equal (read-sreg rname (ISA-SRF (ISA-step (INST-pre-ISA i) intr)))
            (read-sreg rname (ISA-SRF (INST-pre-ISA i)))))
  :hints (("goal" :in-theory (enable  sreg-modifier-p srname-p))))

(in-theory (enable not-member-of-cdr-if-car-is-IFU-stg))

;; End of the lemmas about the behavior of register modifiers in the ISA model.

;; More complex lemmas follows.  Major topics are
;;   pipeline stages and register modifiers.
;;   register modifiers and MA-step.
(encapsulate nil
(local
(defthm not-committed-p-trace-LRM-before
    (implies (and (no-commit-inst-p trace)
                  (trace-exist-uncommitted-LRM-before-p i rname trace))
             (not (committed-p (trace-LRM-before i rname trace))))))


(local
(defthm not-committed-p-LRM-before-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (member-equal i trace)
                  (committed-p
                   (trace-LRM-before i rname trace)))
             (not (trace-exist-uncommitted-LRM-before-p I rname trace)))))

; (exist-LRM-before-p i rname MT) is true only if there exists
; an uncommitted register modifier of register rname before i.
(defthm not-committed-p-LRM-before
     (implies (and (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (INST-in i MT)
                   (committed-p (LRM-before i rname MT)))
              (not (exist-uncommitted-LRM-before-p i rname MT)))
  :hints (("goal" :in-theory (enable LRM-before
                                     INST-in
                                     exist-uncommitted-LRM-before-p)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
        (implies (and (inv MT MA)
                      (MAETT-p MT) (MA-state-p MA)
                      (INST-in i MT)
                      (exist-uncommitted-LRM-before-p I rname MT))
                 (not (committed-p (LRM-before I rname MT)))))))
)

(encapsulate nil
(local
(defthm not-committed-p-trace-LSRM-before
    (implies (and (no-commit-inst-p trace)
                  (trace-exist-uncommitted-LSRM-before-p i rname trace))
             (not (committed-p (trace-LSRM-before i rname trace)))))))
```

```
(local
(defthm not-committed-p-LSRM-before-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (member-equal i trace)
                  (committed-p
                   (trace-LSRM-before i rname trace)))
             (not (trace-exist-uncommitted-LSRM-before-p I rname trace)))))

; exist-LSRM-before-p is true only if there exists an
; uncommitted register modifier of special register rname.
(defthm not-committed-p-LSRM-before
     (implies (and (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (INST-in i MT)
                   (committed-p (LSRM-before I rname MT)))
              (not (exist-uncommitted-LSRM-before-p I rname MT)))
  :hints (("goal" :in-theory (enable LSRM-before
                                     INST-in
                                     exist-uncommitted-LSRM-before-p)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
     (implies (and (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (INST-in i MT)
                   (exist-uncommitted-LSRM-before-p I rname MT))
              (not (committed-p (LSRM-before I rname MT)))))))
)

; Following two lemmas say that last register modifier in the ROB is
; dispatched and not committed.
(encapsulate nil
(local
(defthm dispatched-p-LRM-in-ROB-help
    (implies (trace-exist-LRM-in-ROB-p idx trace)
             (dispatched-p (trace-LRM-in-ROB idx trace)))))

(defthm dispatched-p-LRM-in-ROB
    (implies (exist-LRM-in-ROB-p idx MT)
             (dispatched-p (LRM-in-ROB idx MT)))
  :hints (("goal" :in-theory (enable exist-LRM-in-ROB-p
                                     LRM-in-ROB))))
)

(encapsulate nil
(local
(defthm not-committed-p-LRM-in-ROB-help
    (implies (trace-exist-LRM-in-ROB-p idx trace)
             (not (committed-p (trace-LRM-in-ROB idx trace))))))

(defthm not-committed-p-LRM-in-ROB
    (implies (exist-LRM-in-ROB-p idx MT)
             (not (committed-p (LRM-in-ROB idx MT))))
  :hints (("Goal" :in-theory (enable exist-LRM-in-ROB-p
                                     LRM-in-ROB))))
)
```

```
; (Last-reg-modifier-before i) is a dispatched instruction if
; i is also dispatched.
(defthm dispatched-p-LRM-before-dispatched
    (implies (and (dispatched-p i)
                  (exist-LRM-before-p i rname MT)
                  (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT))
             (dispatched-p (LRM-before i rname MT)))
  :hints (("goal" :use (:instance INST-IN-ORDER-LRM-BEFORE)
                  :in-theory (disable INST-IN-ORDER-LRM-BEFORE))))

; (Last-sreg-modifier-before i) is a dispatched instruction if
; i is also dispatched.
(defthm dispatched-p-LSRM-before-dispatched
    (implies (and (dispatched-p i)
                  (exist-LSRM-before-p i rname MT)
                  (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT))
             (dispatched-p (LSRM-before i rname MT)))
  :hints (("goal" :use (:instance INST-IN-ORDER-LSRM-BEFORE)
                  :in-theory (disable INST-IN-ORDER-LSRM-BEFORE))))

(defthm exist-LRM-before-p-and-exist-uncommitted-LRM-before-p
     (implies (and (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (INST-in i MT))
              (iff (exist-uncommitted-LRM-before-p i rname MT)
                   (and (exist-LRM-before-p i rname MT)
                        (not (committed-p (LRM-before i rname MT)))))))

(defthm exist-LSRM-before-p-and-exist-uncommitted-LSRM-before-p
     (implies (and (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (INST-in i MT))
              (iff (exist-uncommitted-LSRM-before-p i rname MT)
                   (and (exist-LSRM-before-p i rname MT)
                        (not (committed-p (LSRM-before i rname MT)))))))
(in-theory
 (disable exist-LRM-before-p-and-exist-uncommitted-LRM-before-p
          exist-LSRM-before-p-and-exist-uncommitted-LSRM-before-p))

; Following two lemmas say that the last register or special register
; modifier in ROB is not in the dispatch queue.
(encapsulate nil
(local
(defthm not-DQ-stg-p-LRM-in-ROB-help
    (implies (trace-exist-LRM-in-ROB-p idx trace)
             (not (DQ-stg-p (INST-stg (trace-LRM-in-ROB idx trace)))))))

(defthm not-DQ-stg-p-LRM-in-ROB
    (implies (exist-LRM-in-ROB-p idx MT)
             (not (DQ-stg-p (INST-stg (LRM-in-ROB idx MT)))))
  :hints (("Goal" :in-theory (enable LRM-in-ROB exist-LRM-in-ROB-p))))
)

(encapsulate nil
(local
(defthm not-DQ-stg-p-LSRM-in-ROB-help
    (implies (trace-exist-LSRM-in-ROB-p idx trace)
             (not (DQ-stg-p (INST-stg (trace-LSRM-in-ROB idx trace)))))))
```

```
(defthm not-DQ-stg-p-LSRM-in-ROB
    (implies (exist-LSRM-in-ROB-p idx MT)
             (not (DQ-stg-p (INST-stg (LSRM-in-ROB idx MT)))))
  :hints (("Goal" :in-theory (enable LSRM-in-ROB exist-LSRM-in-ROB-p))))
)

(encapsulate nil
(local
(defthm not-trace-exist-LRM-in-ROB-p-if-no-dispatched-inst-p
    (implies (no-dispatched-inst-p trace)
             (not (trace-exist-LRM-in-ROB-p rname trace)))))

; A help lemma.
(defthm not-trace-reg-modifier-cdr-if-car-not-dispatched
    (implies (and (consp trace)
                  (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (not (dispatched-p (car trace))))
             (not (trace-exist-LRM-in-ROB-p rname (cdr trace))))
  :hints (("goal" :cases ((no-dispatched-inst-p (cdr trace))))
          ("subgoal 1" :in-theory (disable NO-DISPATCHED-INST-P-CDR))))
)

(encapsulate nil
(local
(defthm exist-LRM-before-p-undispatched-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-p i) (member-equal i trace)
                  (INST-listp trace)
                  (not (dispatched-p i))
                  (trace-exist-LRM-in-ROB-p rname trace))
             (trace-exist-LRM-before-p i rname trace))))

; If instruction i is not dispatched, and a register modifier of rname is
; in the ROB,  there exist register modifiers preceding i.
(defthm exist-LRM-before-p-undispatched
    (implies (and (inv MT MA)
                  (not (dispatched-p i))
                  (exist-LRM-in-ROB-p rname MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (exist-LRM-before-p i rname MT))
  :hints (("goal" :in-theory (enable exist-LRM-before-p
                                     exist-LRM-in-ROB-p INST-in))))
)

(encapsulate nil
(local
(defthm not-trace-exist-LSRM-in-ROB-p-if-no-dispatched-inst-p
    (implies (no-dispatched-inst-p trace)
             (not (trace-exist-LSRM-in-ROB-p rname trace)))))

; A help lemma.
(defthm not-trace-sreg-modifier-cdr-if-car-not-dispatched
    (implies (and (consp trace)
                  (inv MT MA)
```

```
                (MAETT-p MT)
                (MA-state-p MA)
                (subtrace-p trace MT)
                (INST-listp trace)
                (not (dispatched-p (car trace))))
           (not (trace-exist-LSRM-in-ROB-p rname (cdr trace))))
  :hints (("goal" :cases ((no-dispatched-inst-p (cdr trace))))
          ("subgoal 1" :in-theory (disable NO-DISPATCHED-INST-P-CDR))))
)

(encapsulate nil
(local
(defthm exist-LSRM-before-p-undispatched-help
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (subtrace-p trace MT)
                (INST-p i) (member-equal i trace)
                (INST-listp trace)
                (not (dispatched-p i))
                (trace-exist-LSRM-in-ROB-p rname trace))
           (trace-exist-LSRM-before-p i rname trace))))

; If instruction i is not dispatched, and a register modifier of
; special register rname is in the ROB,  there exist register modifiers
; preceding i.
(defthm exist-LSRM-before-p-undispatched
    (implies (and (inv MT MA)
                (not (dispatched-p i))
                (exist-LSRM-in-ROB-p rname MT)
                (MAETT-p MT) (MA-state-p MA)
                (INST-p i) (INST-in i MT))
           (exist-LSRM-before-p i rname MT))
  :hints (("goal" :in-theory (enable exist-LSRM-before-p
                                     exist-LSRM-in-ROB-p INST-in))))
)

(encapsulate nil
(local
(defthm exist-LRM-in-ROB-p-iff-exist-uncommitted-LRM-before-p-help
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (subtrace-p trace MT)
                (INST-listp trace)
                (equal (INST-stg i) '(DQ 0))
                (INST-p i) (member-equal i trace))
           (equal (trace-exist-uncommitted-LRM-before-p i rname trace)
                  (trace-exist-LRM-in-ROB-p rname trace)))))

; If instruction i is at (DQ 0), (exist-LRM-before-p i rname MT) iff
; (exist-LRM-in-ROB-p rname MT).  See the comment of
; INST-dest-val-LRM-in-ROB
(defthm exist-LRM-in-ROB-p-iff-exist-uncommitted-LRM-before-p
   (implies (and (inv MT MA)
                 (equal (INST-stg i) '(DQ 0))
                 (MAETT-p MT) (MA-state-p MA)
                 (INST-p i) (INST-in i MT))
            (equal (exist-uncommitted-LRM-before-p i rname MT)
                   (exist-LRM-in-ROB-p rname MT)))
  :hints (("goal" :in-theory (enable exist-LRM-in-ROB-p
                                     exist-uncommitted-LRM-before-p
                                     INST-in)))
  :rule-classes
```

```
  ((:rewrite :corollary
             (implies (and (inv MT MA)
                           (exist-LRM-in-ROB-p rname MT)
                           (equal (INST-stg i) '(DQ 0))
                           (MAETT-p MT) (MA-state-p MA)
                           (INST-p i) (INST-in i MT))
                      (exist-uncommitted-LRM-before-p i rname MT)))
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (not (exist-LRM-in-ROB-p rname MT))
                           (equal (INST-stg i) '(DQ 0))
                           (MAETT-p MT) (MA-state-p MA)
                           (INST-p i) (INST-in i MT))
                      (not (exist-uncommitted-LRM-before-p i rname MT))))))

(local
(defthm INST-dest-val-LRM-in-ROB-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (equal (INST-stg i) '(DQ 0))
                  (INST-listp trace)
                  (INST-p i) (member-equal i trace)
                  (trace-exist-LRM-in-ROB-p rname trace))
             (equal (trace-LRM-in-ROB rname trace)
                    (trace-LRM-before i rname trace)))))

; If instruction i is at (DQ 0), the last register modifier before i
; is the last register modifier in ROB.  This is a critical lemma in
; proving the correctness of our invariants.  The register reference table
; keeps track of the last instruction in the ROB that modifies registers.
; A dispatched instruction uses this information to get the operand source
; value.  This lemmas implies that that behavior is correct because
; the source operand for instruction i should come from the last register
; modifier before i.
(defthm INST-dest-val-LRM-in-ROB
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(DQ 0))
                  (exist-LRM-in-ROB-p rname MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (LRM-in-ROB rname MT)
                    (LRM-before i rname MT)))
  :hints (("goal" :in-theory (enable LRM-in-ROB
                                     LRM-before
                                     exist-LRM-in-ROB-p
                                     INST-in))))
)

(encapsulate nil
(local
(defthm exist-LSRM-in-ROB-p-iff-exist-uncommitted-LSRM-before-p-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (equal (INST-stg i) '(DQ 0))
                  (INST-p i) (member-equal i trace))
             (equal (trace-exist-uncommitted-LSRM-before-p i rname trace)
                    (trace-exist-LSRM-in-ROB-p rname trace)))))

; If instruction i is at (DQ 0), (exist-LSRM-before-p i rname MT) and
```

```
; (exist-LSRM-in-ROB-p rname MT) are equivalent.
(defthm exist-LSRM-in-ROB-p-iff-exist-uncommitted-LSRM-before-p
   (implies (and (inv MT MA)
                 (equal (INST-stg i) '(DQ 0))
                 (MAETT-p MT) (MA-state-p MA)
                 (INST-p i) (INST-in i MT))
            (equal (exist-uncommitted-LSRM-before-p i rname MT)
                   (exist-LSRM-in-ROB-p rname MT)))
  :hints (("goal" :in-theory (enable exist-LSRM-in-ROB-p
                                     exist-uncommitted-LSRM-before-p
                                     INST-in)))
  :rule-classes
  ((:rewrite :corollary
             (implies (and (inv MT MA)
                           (equal (INST-stg i) '(DQ 0))
                           (MAETT-p MT) (MA-state-p MA)
                           (INST-p i) (INST-in i MT)
                           (exist-LSRM-in-ROB-p rname MT))
                      (exist-uncommitted-LSRM-before-p i rname MT)))
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (equal (INST-stg i) '(DQ 0))
                           (MAETT-p MT) (MA-state-p MA)
                           (INST-p i) (INST-in i MT)
                           (not (exist-LSRM-in-ROB-p rname MT)))
                      (not (exist-uncommitted-LSRM-before-p i rname MT))))))

(local
(defthm INST-dest-val-LSRM-in-ROB-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (equal (INST-stg i) '(DQ 0))
                  (INST-listp trace)
                  (INST-p i) (member-equal i trace)
                  (trace-exist-LSRM-in-ROB-p rname trace))
             (equal (trace-LSRM-in-ROB rname trace)
                    (trace-LSRM-before i rname trace)))))

; If instruction i is at (DQ 0), the last register modifier before i
; is the last register modifier in ROB.  See the comment
; of INST-dest-val-LRM-in-ROB.
(defthm INST-dest-val-LSRM-in-ROB
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(DQ 0))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (exist-LSRM-in-ROB-p rname MT))
             (equal (LSRM-in-ROB rname MT)
                    (LSRM-before i rname MT)))
    :hints (("goal" :in-theory (enable LSRM-in-ROB
                                       LSRM-before
                                       exist-LSRM-in-ROB-p
                                       INST-in))))
)

;  The ROB contains a field complete?.  This field is 1 if the corresponding
; instruction has completed its execution.  This lemma says that
; the last register modifier is still being executed if the corresponding
; complete? flag in the ROB is off.
(defthm execute-stg-LRM-before
    (implies (and (inv MT MA)
```

```
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (exist-uncommitted-LRM-before-p I rname MT)
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (robe-complete?
                            (nth-robe (INST-tag (LRM-before i rname MT))
                                      (MA-rob MA))))))
     (execute-stg-p (INST-stg (LRM-before i rname MT))))
  :hints (("goal" :in-theory (e/d (IFU-IS-LAST-INST
                                   DQO-is-earlier-than-other-DQ)
                                  (INST-in-order-LRM-before
                                   INST-is-at-one-of-the-stages
                                      committed-p))
                  :use ((:instance INST-is-at-one-of-the-stages
                            (i (LRM-before i rname MT)))
                        (:instance INST-in-order-LRM-before)))))

; Similar to the lemma above.
(defthm execute-stg-LSRM-before
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (exist-uncommitted-LSRM-before-p I rname MT)
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (robe-complete?
                            (nth-robe (INST-tag (LSRM-before i rname MT))
                                      (MA-rob MA))))))
     (execute-stg-p (INST-stg (LSRM-before i rname MT))))
  :hints (("goal" :in-theory (e/d (IFU-IS-LAST-INST
                                   DQO-is-earlier-than-other-DQ)
                                  (INST-in-order-LSRM-before
                                   INST-is-at-one-of-the-stages
                                      committed-p))
                  :use ((:instance INST-is-at-one-of-the-stages
                                   (i (LSRM-before i rname MT)))
                        (:instance INST-in-order-LSRM-before)))))

(encapsulate nil
(local
(defthm not-exist-LRM-before-p-step-INST-help
    (implies (and (inv MT MA)
                  (member-equal i trace) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (subtrace-p trace MT) (INST-listp trace)
                  (not (trace-exist-LRM-before-p i idx trace))
                  (MAETT-p MT) (MA-state-p MA))
             (not (trace-exist-LRM-before-p (step-inst i MT MA sigs)
                      idx (step-trace trace MT MA sigs ISA spc smc))))))

(local
(defthm exist-LRM-before-p-step-INST-help
    (implies (and (inv MT MA)
                  (member-equal i trace) (INST-p i)
                  (subtrace-p trace MT) (INST-listp trace)
                  (not (b1p (flush-all? MA sigs)))
                  (trace-exist-LRM-before-p i idx trace)
                  (MAETT-p MT) (MA-state-p MA))
             (trace-exist-LRM-before-p (step-inst i MT MA sigs)
                      idx (step-trace trace MT MA sigs ISA spc smc)))))

(local
(defthm LRM-before-step-INST-help
```

```
    (Implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (member-equal i trace) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (trace-exist-LRM-before-p i idx trace)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (trace-LRM-before (step-INST i MT MA sigs)
                            idx (step-trace trace MT MA sigs ISA spc smc))
                    (step-INST (trace-LRM-before i idx trace)
                               MT MA sigs)))))

; Commutativity of LRM-before and step-INST.
(defthm LRM-before-step-INST
    (Implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
;                 (execute-stg-p
;                  (INST-stg (LRM-before i idx MT)))
                  (exist-LRM-before-p i idx MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LRM-before (step-INST i MT MA sigs)
                                idx (MT-step MT MA sigs))
                    (step-INST (LRM-before i idx MT)
                               MT MA sigs)))
  :hints (("Goal" :in-theory (enable LRM-before
                                     exist-LRM-before-p INST-in))))

; If there is an register modifier before i, then there still is a register
; modifier before i in the next cycle.
(defthm exist-LRM-before-p-step-INST
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (exist-LRM-before-p i idx MT)
;                 (execute-stg-p (INST-stg
;                                 (LRM-before i idx MT)))
                  (MAETT-p MT) (MA-state-p MA))
             (exist-LRM-before-p (step-INST i MT MA sigs)
                                 idx (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable exist-LRM-before-p
                                     LRM-before
                                     INST-in))))

(local
(defthm not-exist-uncommitted-LRM-before-p-step-INST-help
    (implies (and (inv MT MA)
                  (member-equal i trace) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (subtrace-p trace MT) (INST-listp trace)
                  (not (trace-exist-uncommitted-LRM-before-p i idx trace))
                  (MAETT-p MT) (MA-state-p MA))
             (not (trace-exist-uncommitted-LRM-before-p
                   (step-inst i MT MA sigs)
                   idx (step-trace trace MT MA sigs ISA spc smc))))))

(local
(defthm exist-uncommitted-LRM-before-p-step-INST-help
    (implies (and (inv MT MA)
                  (member-equal i trace) (INST-p i)
                  (subtrace-p trace MT) (INST-listp trace)
                  (not (b1p (flush-all? MA sigs)))
                  (trace-exist-uncommitted-LRM-before-p i idx trace)
```

```
                (execute-stg-p
                 (INST-stg
                  (trace-LRM-before i idx trace)))
                (MAETT-p MT) (MA-state-p MA))
           (trace-exist-uncommitted-LRM-before-p (step-inst i MT MA sigs)
                    idx (step-trace trace MT MA sigs ISA spc smc)))))

(defthm exist-uncommitted-LRM-before-p-step-INST
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (not (b1p (flush-all? MA sigs)))
                (exist-uncommitted-LRM-before-p i idx MT)
                (execute-stg-p (INST-stg
                                (LRM-before i idx MT)))
                (MAETT-p MT) (MA-state-p MA))
           (exist-uncommitted-LRM-before-p (step-INST i MT MA sigs)
                                           idx (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable exist-uncommitted-LRM-before-p
                                     LRM-before
                                     INST-in))))

; The last register modifier continues to have the same ROB entry number
; in the next cycle.
(defthm INST-tag-LRM-before-step-INST
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (not (b1p (flush-all? MA sigs)))
                (exist-LRM-before-p i idx MT)
                (execute-stg-p (INST-stg
                                (LRM-before i idx MT)))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (equal (INST-tag (LRM-before (step-INST i MT MA sigs)
                                        idx (MT-step MT MA sigs)))
                  (INST-tag (LRM-before i idx MT))))
  :hints (("goal" :in-theory (enable exist-LRM-before-p
                                     LRM-before
                                     INST-in))))
)

(encapsulate nil
(local
(defthm not-exist-LSRM-before-p-step-INST-help
    (implies (and (inv MT MA)
                (member-equal i trace) (INST-p i)
                (subtrace-p trace MT) (INST-listp trace)
                (not (b1p (flush-all? MA sigs)))
                (not (trace-exist-LSRM-before-p i idx trace))
                (MAETT-p MT) (MA-state-p MA))
           (not (trace-exist-LSRM-before-p (step-inst i MT MA sigs)
                    idx (step-trace trace MT MA sigs ISA spc smc))))))

(local
(defthm exist-LSRM-before-p-step-INST-help
    (implies (and (inv MT MA)
                (member-equal i trace) (INST-p i)
                (subtrace-p trace MT) (INST-listp trace)
                (not (b1p (flush-all? MA sigs)))
                (trace-exist-LSRM-before-p i idx trace)
                (MAETT-p MT) (MA-state-p MA))
           (trace-exist-LSRM-before-p (step-inst i MT MA sigs)
                    idx (step-trace trace MT MA sigs ISA spc smc)))))
```

```
(local
(defthm LSRM-before-step-INST-help
    (Implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (member-equal i trace) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (trace-exist-LSRM-before-p i idx trace)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (trace-LSRM-before (step-INST i MT MA sigs)
                            idx (step-trace trace MT MA sigs ISA spc smc))
                    (step-INST (trace-LSRM-before i idx trace)
                               MT MA sigs)))))

; Commutativity of LSRM-before and step-INST.
(defthm LSRM-before-step-INST
    (Implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (exist-LSRM-before-p i idx MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSRM-before (step-INST i MT MA sigs)
                                 idx (MT-step MT MA sigs))
                    (step-INST (LSRM-before i idx MT)
                               MT MA sigs)))
  :hints (("Goal" :in-theory (enable LSRM-before
                                     exist-LSRM-before-p INST-in))))

; If special register modifiers exist in the current cycle, they will
; in the next cycle.
(defthm exist-LSRM-before-p-step-INST
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (exist-LSRM-before-p i idx MT)
                  (MAETT-p MT) (MA-state-p MA))
             (exist-LSRM-before-p (step-INST i MT MA sigs)
                                  idx (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable exist-LSRM-before-p
                                     LSRM-before
                                     INST-in))))

(local
(defthm not-exist-uncommitted-LSRM-before-p-step-INST-help
    (implies (and (inv MT MA)
                  (member-equal i trace) (INST-p i)
                  (subtrace-p trace MT) (INST-listp trace)
                  (not (b1p (flush-all? MA sigs)))
                  (not (trace-exist-uncommitted-LSRM-before-p i idx trace))
                  (MAETT-p MT) (MA-state-p MA))
             (not (trace-exist-uncommitted-LSRM-before-p
                   (step-inst i MT MA sigs)
                   idx (step-trace trace MT MA sigs ISA spc smc))))))

(local
(defthm exist-uncommitted-LSRM-before-p-step-INST-help
    (implies (and (inv MT MA)
                  (member-equal i trace) (INST-p i)
                  (subtrace-p trace MT) (INST-listp trace)
                  (not (b1p (flush-all? MA sigs)))
                  (trace-exist-uncommitted-LSRM-before-p i idx trace)
                  (execute-stg-p (INST-stg (trace-LSRM-before i idx trace)))
                  (MAETT-p MT) (MA-state-p MA))
```

```
            (trace-exist-uncommitted-LSRM-before-p (step-inst i MT MA sigs)
                     idx (step-trace trace MT MA sigs ISA spc smc)))))

(defthm exist-uncommitted-LSRM-before-p-step-INST
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (exist-uncommitted-LSRM-before-p i idx MT)
                  (execute-stg-p (INST-stg (LSRM-before i idx MT)))
                  (MAETT-p MT) (MA-state-p MA))
             (exist-uncommitted-LSRM-before-p (step-INST i MT MA sigs)
                                              idx (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable exist-uncommitted-LSRM-before-p
                                     LSRM-before
                                     INST-in))))

; The special register modifier continues to have the same tag entry
; number in the next cycle.
(defthm INST-tag-LSRM-before-step-INST
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (exist-LSRM-before-p i idx MT)
                  (execute-stg-p (INST-stg (LSRM-before i idx MT)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (equal (INST-tag (LSRM-before (step-INST i MT MA sigs)
                                           idx (MT-step MT MA sigs)))
                    (INST-tag (LSRM-before i idx MT))))
  :hints (("goal" :in-theory (enable exist-LSRM-before-p
                                     LSRM-before
                                     INST-in))))
)

;;;;;;;;;;;;;;;;;End of reg-modifier theory;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;Lemmas about memory modifiers;;;;;;;;;;;;;;;;;;;;;;;;
(encapsulate nil
(local
 (defthm inst-in-LMM-before-help
    (implies (trace-exist-LMM-before-p i addr trace)
             (member-equal (trace-LMM-before i addr trace)
                           trace))))

(defthm inst-in-LMM-before
    (implies (exist-LMM-before-p i addr MT)
             (INST-in (LMM-before i addr MT) MT))
  :hints (("goal" :in-theory (enable INST-in LMM-before exist-LMM-before-p))))
)

(encapsulate nil
(local
 (defthm LMM-before-differs-from-i-help
    (implies (trace-exist-LMM-before-p i addr MT)
             (not (equal (trace-LMM-before i addr MT) i)))))

; The last memory modifier before i is not i.
(defthm LMM-before-differs-from-i
    (implies (exist-LMM-before-p i addr MT)
             (not (equal (LMM-before i addr MT) i)))
  :hints (("goal" :in-theory (enable exist-LMM-before-p
                                     LMM-before))))
)
```

```
(encapsulate nil
(local
 (defthm INST-in-order-LMM-before-help
    (implies (and (member-equal i trace)
                  (trace-exist-LMM-before-p I addr trace))
             (member-in-order (trace-LMM-before i addr trace)
                              i trace))
   :hints (("goal" :in-theory (enable member-in-order*)))))

; The last memory modifier before i precedes i.
(defthm INST-in-order-LMM-before
    (implies (and (INST-in i MT) (exist-LMM-before-p I addr MT))
             (INST-in-order-p (LMM-before i addr MT) i MT))
  :hints (("goal" :in-theory (enable LMM-before INST-in
                                     INST-in-order-p exist-LMM-before-p))))
)

(encapsulate nil
(local
(defthm mem-modifier-p-LMM-before-help
    (implies (trace-exist-LMM-before-p I addr trace)
             (mem-modifier-p addr (trace-LMM-before I addr trace)))))

; The last memory modifier is a memory modifier.
(defthm mem-modifier-p-LMM-before
    (implies (exist-LMM-before-p I addr MT)
             (mem-modifier-p addr (LMM-before I addr MT)))
  :hints (("goal" :in-theory (enable exist-LMM-before-p LMM-before))))
)

; A mem-modifier satisfies INST-store?
(defthm not-mem-modifier-p-if-not-INST-store?
    (implies (not (b1p (INST-store? i)))
             (not (mem-modifier-p addr i)))
  :hints (("goal" :in-theory (enable mem-modifier-p
                                     INST-function-def
                                     decode logbit* rdb lift-b-ops))))

; A last memory modifier satisfies INST-store?
(defthm INST-store-LRM-before
    (implies (and (MAETT-p MT) (INST-p i)
                  (exist-LMM-before-p I addr MT))
             (equal (INST-store? (LMM-before I addr MT))
                    1))
  :hints (("goal" :in-theory (e/d (equal-b1p-converter)
                                  (not-mem-modifier-p-if-not-INST-store?))
                  :use (:instance not-mem-modifier-p-if-not-INST-store?
                                  (i (LMM-before I addr MT))))))

; If the access address of an last memory modifier at address addr is, in
; fact, addr.
(defthm INST-store-addr-LMM
    (implies (exist-LMM-before-p i addr MT)
             (equal (INST-store-addr (LMM-before i addr MT))
                    addr))
  :hints (("goal" :in-theory (e/d (mem-modifier-p)
                                  (MEM-MODIFIER-P-LMM-BEFORE))
                  :use (:instance MEM-MODIFIER-P-LMM-BEFORE))))

; (Last-mem-modifier-before i) is a dispatched instruction if
; i is dispatched, because instructions are dispatched in program order.
```

```
(defthm dispatched-p-LMM-before-dispatched
    (implies (and (dispatched-p i)
                  (exist-LMM-before-p i addr MT)
                  (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT))
             (dispatched-p (LMM-before i addr MT)))
  :hints (("goal" :use (:instance INST-IN-ORDER-LMM-BEFORE)
                  :in-theory (disable INST-IN-ORDER-LMM-BEFORE))))

; If j is the last register modifier before i, and i is not speculatively
; executed instruction, then j is not speculatively executed, either.
(defthm INST-specultv-LMM-before
    (implies (and (inv MT MA)
                  (exist-LMM-before-p i addr MT)
                  (not (b1p (INST-specultv? i)))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (INST-specultv? (LMM-before i addr MT))
                    0))
  :hints (("goal" :in-theory (e/d (equal-b1p-converter)
                                  (INST-in-order-p-INST-specultv))
                  :use (:instance INST-in-order-p-INST-specultv
                                  (i (LMM-before i addr MT))
                                  (j i)))))

; If j is the last register modifier before i, and i's modified flag is not
; on, then j's modified flag is not on either.
(defthm INST-modified-LMM-before
    (implies (and (inv MT MA)
                  (exist-LMM-before-p i addr MT)
                  (not (b1p (INST-modified? I)))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (INST-modified? (LMM-before i addr MT))
                    0))
  :hints (("goal" :in-theory (e/d (equal-b1p-converter)
                                  (INST-in-order-p-INST-modified))
                  :use (:instance INST-in-order-p-INST-modified
                                  (i (LMM-before i addr MT))
                                  (j i)))))

(encapsulate nil
(local
 (defthm not-fetch-error-detected-p-if-commit-stg-p
     (implies (and (inv MT MA)
                   (INST-in i MT) (INST-p i)
                   (MAETT-p MT) (MA-state-p MA)
                   (not (b1p (INST-modified? i)))
                   (commit-stg-p (INST-stg i)))
              (not (INST-fetch-error-detected-p i)))
   :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter
                                    NOT-INST-SPECULTV-INST-IN-IF-COMMITTED
                                                  INST-excpt-detected-p)
                                   (INST-inv-if-INST-in))
                   :use (:instance INST-inv-if-INST-in)))))


; If j is the last register modifier before i, and i is not speculatively
; executed, then no fetch error is detected for j.  (In our theory,
```

```
; speculation includes the execution of all instructions that should not
; be executed by the ISA.  Thus, an instruction following an exception
; is a speculatively executed instruction.  )
(defthm INST-fetch-error-detected-p-LMM-before
    (implies (and (inv MT MA)
                  (exist-LMM-before-p i addr MT)
                  (not (retire-stg-p (INST-stg (LMM-before i addr MT))))
                  (not (b1p (INST-specultv? I)))
                  (not (b1p (INST-modified? I)))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (not (INST-fetch-error-detected-p (LMM-before i addr MT))))
    :hints (("goal" :in-theory (e/d (equal-b1p-converter
                                     INST-start-specultv?
                                     committed-p
                                     INST-excpt?
                                     lift-b-ops)
                                    (INST-in-order-p-INST-start-specultv))
                    :use (:instance INST-in-order-p-INST-start-specultv
                                    (i (LMM-before i addr MT))
                                    (j i))
                    :restrict ((not-fetch-error-detected-p-if-commit-stg-p
                                ((i (LMM-before i addr MT))))))))
)

(encapsulate nil
(local
 (defthm not-excpt-detected-p-if-commit-stg-p
     (implies (and (inv MT MA)
                   (INST-in i MT) (INST-p i)
                   (MAETT-p MT) (MA-state-p MA)
                   (not (b1p (INST-modified? i)))
                   (commit-stg-p (INST-stg i)))
              (not (INST-excpt-detected-p i)))
   :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter
                                    NOT-INST-SPECULTV-INST-IN-IF-COMMITTED)
                                   (INST-inv-if-INST-in))
                   :use (:instance INST-inv-if-INST-in)))))

; If j is the last register modifier before i in program order, and
; i is not speculatively executed, then j has not raised an exception.
; This is because the MAETT records instructions following an exceptions
; as a speculatively executed instruction.
(defthm INST-excpt-detected-p-LMM-before
    (implies (and (inv MT MA)
                  (exist-LMM-before-p i addr MT)
                  (not (retire-stg-p (INST-stg (LMM-before i addr MT))))
                  (not (b1p (INST-specultv? I)))
                  (not (b1p (INST-modified? I)))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (not (INST-excpt-detected-p (LMM-before i addr MT))))
    :hints (("goal" :in-theory (e/d (equal-b1p-converter
                                     INST-start-specultv?
                                     committed-p
                                     INST-excpt?
                                     lift-b-ops)
                                    (INST-in-order-p-INST-start-specultv))
                    :use (:instance INST-in-order-p-INST-start-specultv
                                    (i (LMM-before i addr MT))
```

```
                                  (j i)))))
)

(encapsulate nil
(local
(defthm not-excpt-detected-p-if-commit-stg-p
     (implies (and (inv MT MA)
                   (INST-in i MT) (INST-p i)
                   (MAETT-p MT) (MA-state-p MA)
                   (not (b1p (INST-modified? i)))
                   (commit-stg-p (INST-stg i)))
              (not (b1p (INST-excpt? i))))
   :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter
                                    NOT-INST-SPECULTV-INST-IN-IF-COMMITTED)
                                     (INST-inv-if-INST-in))
                   :use (:instance INST-inv-if-INST-in)))))

; See the comment about INST-excpt-detected-p-LMM-before.
(defthm INST-excpt-LMM-before
    (implies (and (inv MT MA)
                  (exist-LMM-before-p i addr MT)
                  (not (retire-stg-p (INST-stg (LMM-before i addr MT))))
                  (not (b1p (INST-specultv? I)))
                  (not (b1p (INST-modified? I)))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (INST-excpt? (LMM-before i addr MT)) 0))
  :hints (("goal" :in-theory (e/d (equal-b1p-converter
                                   INST-start-specultv?
                                   committed-p
                                   lift-b-ops)
                                  (INST-in-order-p-INST-start-specultv))
                  :use (:instance INST-in-order-p-INST-start-specultv
                                  (i (LMM-before i addr MT))
                                  (j i)))))
)

(encapsulate nil
(local
(defthm  LMM-is-last-help-help
    (implies (trace-exist-LMM-before-p i addr trace)
             (member-equal (trace-LMM-before i addr trace)
                           trace))))

(local
(defthm LMM-is-last-help
    (implies (and (distinct-member-p trace)
                  (member-equal j trace)
                  (mem-modifier-p addr j)
                  (member-in-order j i trace)
                  (not (equal (trace-LMM-before i addr trace) j)))
             (member-in-order j (trace-LMM-before i addr trace)
                              trace))
  :hints (("goal" :in-theory (enable member-in-order*)))))

; If j is a memory modifier at address addr, and j precedes i in program
; order, then j is the last memory modifier at addr, or the last memory
; modifier exists between j and i.
(defthm LMM-is-last
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
```

```
                (INST-in j MT) (INST-p j)
                (mem-modifier-p addr j)
                (INST-in i MT) (INST-p i)
                (INST-in-order-p j i MT)
                (not (equal (LMM-before i addr MT) j)))
           (INST-in-order-p j (LMM-before i addr MT) MT))
  :hints (("goal" :in-theory (enable INST-in-order-p LMM-before
                                     MT-distinct-inst-p
                                     inv weak-inv
                                     INST-in))))
)

(encapsulate nil
(local
(defthm exist-LMM-if-exist-non-retired-LMM-help
    (implies (trace-exist-non-retired-LMM-before-p i addr trace)
             (trace-exist-LMM-before-p i addr trace))))

; Non-retired memory modifier is a memory modifier that has not
; retired.  If there exists an non-retired memory modifier before i,
; there is, in the normal sense, a memory modifier (the non-retired
; memory modifier itself).
(defthm exist-LMM-if-exist-non-retired-LMM
    (implies (exist-non-retired-LMM-before-p i addr MT)
             (exist-LMM-before-p i addr MT))
  :hints (("goal" :in-theory (enable exist-non-retired-LMM-before-p
                                     exist-LMM-before-p))))

)

; Instruction at write-buffer wbuf0 is a memory modifier at address
; (INST-store-addr i).
(defthm mem-modifier-p-INST-store-addr-if-wbuf0-stg
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (wbuf0-stg-p (INST-stg i)))
             (mem-modifier-p (INST-store-addr i) i))
  :hints (("goal" :in-theory (enable wbuf0-stg-p mem-modifier-p
                                     LSU-STORE-IF-AT-LSU-WBUF))))

; Instruction at write-buffer wbuf1 is a memory modifier at address
; (INST-store-addr i).
(defthm mem-modifier-p-INST-store-addr-if-wbuf1-stg
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (wbuf1-stg-p (INST-stg i)))
             (mem-modifier-p (INST-store-addr i) i))
  :hints (("goal" :in-theory (enable wbuf1-stg-p mem-modifier-p
                                     LSU-STORE-IF-AT-LSU-WBUF))))

; Following several lemmas help to locate the stage of the last memory
; modifier.  The last memory modifier before i is not at IFU stage or DQ
; stage when i is already in the execution stage, because the last memory
; modifier precedes i in program order.  The memory modifier cannot be
; in the integer unit, multiplier unit, or branch unit, of course.
(defthm not-IFU-stg-p-LMM-before
```

```
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (execute-stg-p (INST-stg i))
                  (exist-LMM-before-p i addr MT))
             (not (IFU-stg-p (INST-stg (LMM-before i addr MT)))))
  :hints (("goal" :use (:instance INST-in-order-LMM-before)
                  :in-theory (disable
                              INST-in-order-LMM-before
                      DISPATCHED-P-LMM-BEFORE-DISPATCHED))))

(defthm not-DQ-stg-p-LMM-before
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (execute-stg-p (INST-stg i))
                  (exist-LMM-before-p i addr MT))
             (not (DQ-stg-p (INST-stg (LMM-before i addr MT)))))
  :hints (("goal" :in-theory (e/d (dispatched-p)
                                  (DISPATCHED-P-LMM-BEFORE-DISPATCHED))
                  :use (:instance DISPATCHED-P-LMM-BEFORE-DISPATCHED))))

(defthm not-IU-stg-p-LMM-before
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (exist-LMM-before-p i addr MT)
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (not (IU-stg-p (INST-stg (LMM-before i addr MT)))))
  :hints (("goal" :cases ((b1p (INST-store? (LMM-before i addr MT)))))
          ("subgoal 1" :in-theory (e/d (lift-b-ops equal-b1p-converter)
                                       (INST-STORE-LRM-BEFORE
                                        INST-IU-IF-IU-STG-P))
                       :use (:instance INST-IU-IF-IU-STG-P
                                       (i (LMM-before i addr MT))))))

(defthm not-MU-stg-p-LMM-before
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (exist-LMM-before-p i addr MT)
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (not (MU-stg-p (INST-stg (LMM-before i addr MT)))))
  :hints (("goal" :cases ((b1p (INST-store? (LMM-before i addr MT)))))
          ("subgoal 1" :in-theory (e/d (lift-b-ops equal-b1p-converter)
                                       (INST-STORE-LRM-BEFORE
                                        INST-MU-IF-MU-STG-P))
                       :use (:instance INST-MU-IF-MU-STG-P
                                       (i (LMM-before i addr MT))))))

(defthm not-BU-stg-p-LMM-before
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (exist-LMM-before-p i addr MT)
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (not (BU-stg-p (INST-stg (LMM-before i addr MT)))))
  :hints (("goal" :cases ((b1p (INST-store? (LMM-before i addr MT)))))
          ("subgoal 1" :in-theory (e/d (lift-b-ops equal-b1p-converter)
```

```
                                        (INST-STORE-LRM-BEFORE
                                         INST-BU-IF-BU-STG-P))
                     :use (:instance INST-BU-IF-BU-STG-P
                                     (i (LMM-before i addr MT))))))

; Following several rules specify which stage is not a legal stage for an
; memory modifier to stay.  See invariants in-order-LSU-inst-p.
(encapsulate nil
(local
(defthm not-no-issued-LSU-inst-p-if-member-equal
    (implies (and (member-equal i trace)
                  (LSU-issued-stg-p (INST-stg i)))
             (not (no-issued-LSU-inst-p trace)))))

(local
(defthm LSU-rbuf-RS-in-order-help
    (implies (and (in-order-LSU-issue-p trace)
                  (member-equal i trace) (INST-p i)
                  (equal (INST-stg i) '(LSU rbuf))
                  (member-equal j trace) (INST-p j)
                  (or (equal (INST-stg j) '(LSU RS0))
                      (equal (INST-stg j) '(LSU RS1))))
             (member-in-order i j trace))
  :hints (("goal" :in-theory (enable member-in-order*)))))

(local
(defthm LSU-rbuf-RS-in-order
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU rbuf))
                  (INST-in j MT) (INST-p j)
                  (or (equal (INST-stg j) '(LSU RS0))
                      (equal (INST-stg j) '(LSU RS1)))
                  (MAETT-p MT) (MA-state-p MA))
             (INST-in-order-p i j MT))
  :hints (("goal" :in-theory (enable INST-in-order-p INST-in
                                     inv in-order-LSU-inst-p)))
  :rule-classes nil))

(defthm not-LSU-RS0-LMM-before
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (exist-LMM-before-p i addr MT)
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (not (equal (INST-stg (LMM-before i addr MT))
                         '(LSU RS0))))
  :hints (("goal" :use (:instance LSU-RBUF-RS-IN-ORDER
                                  (j (LMM-before i addr MT))))))

(defthm not-LSU-RS1-LMM-before
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (exist-LMM-before-p i addr MT)
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (not (equal (INST-stg (LMM-before i addr MT))
                         '(LSU RS1))))
```

```
  :hints (("goal" :use (:instance LSU-RBUF-RS-IN-ORDER
                                  (j (LMM-before i addr MT))))))
)

(defthm not-LSU-rbuf-LMM-before
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (exist-LMM-before-p i addr MT)
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (not (equal (INST-stg (LMM-before i addr MT))
                         '(LSU rbuf))))
  :hints (("goal" :cases ((b1p (INST-store? (LMM-before i addr MT)))))
          ("subgoal 1" :in-theory (e/d (lift-b-ops equal-b1p-converter)
                                       (INST-STORE-LRM-BEFORE
                                        LSU-LOAD-IF-AT-LSU-RBUF-LCH))
                       :use (:instance LSU-LOAD-IF-AT-LSU-RBUF-LCH
                                       (i (LMM-before i addr MT))))))

(defthm not-LSU-lch-LMM-before
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (exist-LMM-before-p i addr MT)
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (not (equal (INST-stg (LMM-before i addr MT))
                         '(LSU lch))))
    :hints (("goal" :cases ((b1p (INST-store? (LMM-before i addr MT)))))
          ("subgoal 1" :in-theory (e/d (lift-b-ops equal-b1p-converter)
                                       (INST-STORE-LRM-BEFORE
                                        LSU-LOAD-IF-AT-LSU-RBUF-LCH))
                       :use (:instance LSU-LOAD-IF-AT-LSU-RBUF-LCH
                                       (i (LMM-before i addr MT))))))

(defthm not-complete-normal-LMM-before
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (exist-LMM-before-p i addr MT)
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (not (equal (INST-stg (LMM-before i addr MT))
                         '(complete))))
  :hints (("goal" :use (:instance not-INST-store-if-complete
                                  (i (LMM-before i addr MT)))
                  :in-theory (enable lift-b-ops equal-b1p-converter
                                     exception-relations))))

; If i is not a memory modifier, it does not change the value of memory.
(defthm read-mem-ISA-step-if-not-mem-modifier
    (implies (and (inv MT MA)
                  (INST-in i MT) (addr-p addr) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (mem-modifier-p addr i)))
             (equal (read-mem addr (ISA-mem (ISA-step (INST-pre-ISA i) intr)))
                    (read-mem addr (ISA-mem (INST-pre-ISA i)))))
  :hints (("goal" :in-theory (enable mem-modifier-p INST-store?
                                     ISA-store-inst-p INST-store?
                                     store-inst-p inst-ld-st? INST-LSU?
                                     INST-opcode
```

```
                                    INST-cntlv
                                    lift-b-ops)
                  :use (:instance read-mem-ISA-step-if-not-ISA-store-INST-p
                                  (ISA (INST-pre-ISA i))))))

(encapsulate nil
(local
 (defthm read-mem-LMM-before-help-help
     (implies (and (inv MT MA)
                   (member-equal i trace)
                   (subtrace-p trace MT)
                   (consp trace)
                   (INST-listp trace) (INST-p i)
                   (addr-p addr)
                   (not (trace-exist-LMM-before-p i addr trace))
                   (MAETT-p MT) (MA-state-p MA))
              (equal (read-mem addr (ISA-mem (INST-pre-ISA i)))
                     (read-mem addr (ISA-mem (INST-pre-ISA (car trace))))))
   :hints  ((when-found (member-equal i (cdr trace))
                        (:cases ((consp (cdr trace))))))))


(local
(defthm read-mem-LMM-before-help
    (implies (and  (inv MT MA)
                   (member-equal i trace)
                   (INST-listp trace) (INST-p i)
                   (MAETT-p MT) (MA-state-p MA)
                   (addr-p addr)
                   (subtrace-p trace MT)
                   (trace-exist-LMM-before-p i addr trace))
     (equal (read-mem addr
                      (ISA-mem
                       (INST-post-ISA (trace-LMM-before i addr trace))))
            (read-mem addr (ISA-mem (INST-pre-ISA i)))))
   :hints ((when-found (member-equal i (cdr trace))
                       (:cases ((consp (cdr trace))))))))


; This is a critical lemma.  The memory value at address addr in
; the pre-ISA of i is the same as in the post-ISA of the last memory
; modifier.  In other words, no instruction modifies the value of the memory
; between the last memory modifier before i and i itself.
(defthm read-mem-LMM-before
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (addr-p addr)
                  (MAETT-p MT) (MA-state-p MA)
                  (exist-LMM-before-p i addr MT))
     (equal (read-mem addr
                      (ISA-mem (INST-post-ISA (LMM-before i addr MT))))
            (read-mem addr (ISA-mem (INST-pre-ISA i)))))
  :hints (("goal" :in-theory (enable LMM-before
                                     exist-LMM-before-p
                                     INST-in)))
  :rule-classes nil)
)

;;;;;;;;;;;;;;;;End of memory modifier theory;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

D.6.3 wk-inv.lisp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MI-inv.lisp
; Author  Jun Sawada, University of Texas at Austin
;
;  This book proves the weak invariant wk-inv.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(in-package "ACL2")

(include-book "MA2-lemmas")
(include-book "MAETT-lemmas")

(deflabel begin-wk-inv)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Invariant proof of MT-new-ID-distinct-p MT-distinct-IDs-p
; and MT-distinct-INST-p
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defthm MT-new-ID-distinct-p-init-MT
    (implies (MA-state-p MA)
             (MT-new-ID-distinct-p (init-MT MA)))
  :hints (("Goal" :in-theory (enable init-MT MT-new-ID-distinct-p))))

(defthm ID-lt-all-p-step-trace
    (implies (ID-lt-all-p trace (MT-new-ID MT))
             (ID-lt-all-p (step-trace trace MT MA sigs ISA spc smc)
                          (1+ (MT-new-ID MT))))
  :hints (("Goal" :in-theory (enable exintr-INST fetched-inst))))

(defthm MT-new-ID-distinct-p-MT-step
    (implies (MT-new-ID-distinct-p MT)
             (MT-new-ID-distinct-p (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable MT-new-ID-distinct-p MT-step))))

(defthm MT-distinct-IDs-p-init-MT
    (implies (MA-state-p MA)
             (MT-distinct-IDs-p (init-MT MA)))
  :hints (("Goal" :in-theory (enable init-MT MT-distinct-IDs-p))))

(local
(defthm not-member-eq-ID-step-trace
    (implies (and (weak-inv MT)
                  (subtrace-p trace MT)
                  (INST-in i MT)
                  (not (member-eq-ID i trace)))
             (not (member-eq-ID (step-INST i MT MA sigs)
                                (step-trace trace MT MA sigs
                                            ISA spc smc))))
  :hints (("goal" :in-theory (enable exintr-INST fetched-inst)))))

(local
(defthm distinct-IDs-p-step-trace
    (implies (and (weak-inv MT)
                  (subtrace-p trace MT)
                  (distinct-IDs-p trace))
             (distinct-IDs-p (step-trace trace MT MA sigs ISA spc smc)))))

(defthm MT-distinct-IDs-p-MT-step
    (implies (weak-inv MT)
             (MT-distinct-IDs-p (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable MT-distinct-IDs-p MT-step
                                     weak-inv))))
```

```
(defthm MT-distinct-INST-p-init-MT
    (implies (MA-state-p MA)
             (MT-distinct-INST-p (init-MT MA)))
  :hints (("Goal" :in-theory (enable MT-distinct-INST-p init-MT))))

(local
(encapsulate nil
(local
(defthm distinct-member-p-if-distinct-IDs-p
    (implies (distinct-IDs-p trace)
             (distinct-member-p trace))))

(defthm MT-distinct-INST-p-if-MT-distinct-IDs-p
    (implies (MT-distinct-IDs-p MT)
             (MT-distinct-INST-p MT))
  :hints (("Goal" :in-theory (enable MT-distinct-INST-p MT-distinct-IDs-p))))
))

(defthm MT-distinct-INST-p-MT-step
    (implies (weak-inv MT)
             (MT-distinct-INST-p (MT-step MT MA sigs))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Invariant proof of ISA-step-chain-p
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defthm ISA-step-chain-p-init-MT
    (implies (MA-state-p MA)
             (ISA-step-chain-p (init-MT MA)))
  :hints (("Goal" :in-theory (enable init-MT ISA-step-chain-p))))

(local
(defthm ISA-chained-trace-p-step-trace
    (implies (ISA-chained-trace-p trace ISA)
             (ISA-chained-trace-p (step-trace trace MT MA sigs ISA spc smc)
                                  ISA))
  :hints (("goal" :in-theory (enable exintr-INST fetched-inst)))))

(defthm ISA-step-chain-p-step
    (implies (ISA-step-chain-p MT)
             (ISA-step-chain-p (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable MT-step ISA-step-chain-p))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Invariant Proof of correct-modified-flgs-p
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defthm correct-modified-flgs-p-init-MT
    (implies (MA-state-p MA)
             (correct-modified-flgs-p (init-MT MA)))
  :hints (("goal" :in-theory (enable correct-modified-flgs-p init-MT))))

(local
(encapsulate nil
(local
(defthm MT-modify-p-step-INST-help
    (implies (and (weak-inv MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace2 MT)
                  (INST-listp trace2)
                  (subtrace-p trace MT)
                  (tail-p trace trace2)
```

```
                (consp trace)
                (INST-listp trace)
                (MA-input-p sigs)
                (trace-no-jmp-exintr-before-trace trace trace2 MT MA sigs))
           (equal (trace-modify-p (step-INST (car trace) MT MA sigs)
                                  (step-trace trace2 MT MA sigs
                                              ISA spc smc))
                  (trace-modify-p (car trace) trace2)))
  :hints (("goal" :in-theory (enable INST-MODIFY-P))
          (when-found (step-INST (CAR TRACE) MT MA SIGS)
                      (:expand ((STEP-TRACE TRACE MT MA SIGS ISA SPC SMC)))))))

(defthm MT-modify-p-step-INST
    (implies (and (weak-inv MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (consp trace)
                  (INST-listp trace)
                  (MA-input-p sigs)
                  (MT-no-jmp-exintr-before-trace trace MT MA sigs))
             (equal (MT-modify-p (step-INST (car trace) MT MA sigs)
                                 (MT-step MT MA sigs))
                    (MT-modify-p (car trace) MT)))
  :hints (("Goal" :in-theory (enable MT-modify-p MT-step
                                     MT-no-jmp-exintr-before-trace
                                     subtrace-p
                                     INST-in)
                  :restrict ((MT-modify-p-step-INST-help
                              ((trace2 (MT-trace MT))))))))
))

(local
(encapsulate nil
(local
(defthm local-help
    (implies (and (weak-inv MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (MA-input-p sigs)
                  (trace-no-jmp-exintr-before-trace nil trace MT MA sigs))
      (equal (trace-modify-p (fetched-inst MT
                                           (MT-final-ISA MT)
                                           (specultv-before? nil MT)
                                           (MT-self-modify? MT))
                             (step-trace trace MT MA sigs
                                         (ISA-before trace MT)
                                         (specultv-before? trace MT)
                                         (modified-inst-before? trace MT)))
             (not (trace-no-write-at (ISA-pc (MT-final-ISA MT)) trace))))
  :hints (("goal" :in-theory (enable INST-MODIFY-P)))))

(defthm not-MT-modify-p-if-MT-no-write-at
    (implies (and (weak-inv MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (MT-no-jmp-exintr-before-trace nil MT MA sigs))
     (equal (MT-modify-p (fetched-inst MT (MT-final-ISA MT)
                                       (specultv-before? nil MT)
                                       (MT-self-modify? MT))
                         (MT-step MT MA sigs))
            (not (MT-no-write-at (ISA-pc (MT-final-ISA MT)) MT))))
```

```
  :hints (("Goal" :in-theory (enable MT-modify-p MT-no-write-at
                                     MT-no-jmp-exintr-before-trace)
                  :use (:instance local-help
                                  (trace (MT-trace MT))))))
))


(local
(defthm trace-correct-modified-flgs-p-step-trace
    (implies (and (weak-inv MT)
                  (INST-listp trace)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (INST-listp trace)
                  (subtrace-p trace MT)
                  (MA-input-p sigs)
                  (MT-no-jmp-exintr-before-trace trace MT MA sigs)
                  (trace-correct-modified-flgs-p trace MT
                                        (modified-inst-before? trace MT)))
   (trace-correct-modified-flgs-p (step-trace trace MT MA sigs
                                              (ISA-before trace MT)
                                              (specultv-before? trace MT)
                                              (modified-inst-before? trace MT))
                                  (MT-step MT MA sigs)
                                  (modified-inst-before? trace MT)))
  :hints (("goal" :induct (step-trace trace MT MA sigs ISA
                                                  spc smc))
          (when-pattern-found
           (TRACE-CORRECT-modified-flgs-P (CDR TRACE) MT (@ smc))
           (:expand ((trace-correct-modified-flgs-p trace MT
                                  (modified-inst-before? trace MT)))))
          (when-pattern-found
           (STEP-TRACE (CDR TRACE)
                       (@ MT) (@ ma) (@ sigs)
                       (@ isa) (@ spc) (@ smc))
           (:expand (STEP-TRACE TRACE MT MA SIGS (ISA-BEFORE TRACE MT)
                                (SPECULTV-BEFORE? TRACE MT)
                                (Modified-inst-BEFORE? TRACE MT)))))))

(defthm correct-modified-flgs-p-MT-step
    (implies (and (weak-inv MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs))
             (correct-modified-flgs-p (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable weak-inv
                                     correct-modified-flgs-p)
                  :USE (:INSTANCE trace-correct-modified-flgs-p-step-trace
                                  (trace (MT-trace MT))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Proof of correct-modified-first-preserved
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Proof of correct-modified-first for initial states
(defthm correct-modified-first-init-MT
    (correct-modified-first (init-MT MA))
  :hints (("goal" :in-theory (enable init-MT correct-modified-first))))

;;; invariant proof
(encapsulate nil
(local
(defthm correct-modified-first-preserved-help
    (implies (and (trace-correct-modified-first trace)
```

```
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA))
           (trace-correct-modified-first (step-trace trace MT MA sigs
                                                    ISA spc smc)))
  :hints (("Goal" :in-theory (enable lift-b-ops)))))

(defthm correct-modified-first-preserved
    (implies (and (correct-modified-first MT)
                (MAETT-p MT) (MA-state-p MA))
           (correct-modified-first (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable correct-modified-first
                                     inv correct-modified-first))))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Final theorems for weak-inv
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Weak invariants are true for the initial MAETT.
(defthm weak-inv-init-MT
    (implies (MA-state-p MA)
           (weak-inv (init-MT MA)))
  :hints (("goal" :in-theory (enable weak-inv))))

; Weak invariants is held during MAETT update.
(defthm weak-inv-step
    (implies (and (weak-inv MT)
                (MAETT-p MT)
                (MA-state-p MA)
                (MA-input-p sigs))
           (weak-inv (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable weak-inv))))

(deflabel end-wk-inv)
```

D.6.4 in-order.lisp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; in-order.lisp
; Author  Jun Sawada, University of Texas at Austin
;
;  This book contains the proof of the invariant properties
;  in-order-DQ-p, in-order-LSU-inst-p, and in-order-ROB-p.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(in-package "ACL2")

(include-book "MA2-lemmas")
(include-book "MAETT-lemmas")

(deflabel begin-in-order)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Proof about in-order-DQ-p
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Proof of in-order-DQ-p for initial states.
(defthm in-order-DQ-p-init-MT
    (in-order-DQ-p (init-MT MA))
  :hints (("goal" :in-theory (enable in-order-DQ-p init-MT))))

;;; Proof of in-order-dq-p-preserved.
(defthm INST-stg-step-inst-DQ-if-not-dispatch-inst
```

```
    (implies (and (inv MT MA)
                  (DQ-stg-p (INST-stg i))
                  (not (b1p (dispatch-inst? MA)))
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-stg (step-inst i MT MA sigs)) (INST-stg i)))
  :hints (("Goal" :in-theory (enable step-inst-dq-inst
                                     step-inst-low-level-functions))))

(defthm DQ-stg-idx-new-DQ-stage
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (dispatch-inst? MA)))
                  (not (b1p (DQ-full? (MA-DQ MA)))))
             (equal (DQ-stg-idx (new-DQ-stage MT MA))
                    (MT-DQ-len MT)))
  :hints (("Goal" :in-theory (enable new-DQ-stage DQ-STG-IDX
                                     DQ-full? coerce-dq-stg))))

(defun last-DQ-index-before (sub trace count)
  (if (endp trace)
      count
      (if (equal sub trace)
          count
          (if (DQ-stg-p (INST-stg (car trace)))
              (last-DQ-index-before sub (cdr trace)
                                    (1+ (DQ-stg-idx (INST-stg (car trace)))))
              (last-DQ-index-before sub (cdr trace) count)))))

(defun MT-last-DQ-index-before (sub MT)
  (last-DQ-index-before sub (MT-trace MT) 0))

(in-theory (disable MT-last-DQ-index-before))

(encapsulate nil
(local
(defthm MT-last-DQ-index-before-cdr-if-not-DQ-help
    (implies (and (consp sub) (not (dq-stg-p (INST-stg (car sub))))
                  (tail-p sub trace))
             (equal (last-DQ-index-before (cdr sub) trace count)
                    (last-DQ-index-before sub trace count)))))

(defthm MT-last-DQ-index-before-cdr-if-not-DQ
    (implies (and (consp trace) (not (dq-stg-p (INST-stg (car trace))))
                  (subtrace-p trace MT))
             (equal (MT-last-DQ-index-before (cdr trace) MT)
                    (MT-last-DQ-index-before trace MT)))
  :hints (("Goal" :in-theory (enable MT-last-DQ-index-before
                                     subtrace-p))))
)

(encapsulate nil
(local
(defthm MT-last-DQ-index-before-cdr-if-DQ-help
    (implies (and (consp sub) (dq-stg-p (INST-stg (car sub)))
                  (tail-p sub trace))
             (equal (last-DQ-index-before (cdr sub) trace count)
                    (1+ (DQ-stg-idx (INST-stg (car sub))))))))

(defthm MT-last-DQ-index-before-cdr-if-DQ
    (implies (and (consp trace) (dq-stg-p (INST-stg (car trace)))
                  (subtrace-p trace MT))
```

```
            (equal (MT-last-DQ-index-before (cdr trace) MT)
                   (1+ (DQ-stg-idx (INST-stg (car trace))))))
  :hints (("Goal" :in-theory (enable MT-last-DQ-index-before
                                     subtrace-p))))
)

(defthm MT-DQ-len-non-zero-if-DQ-inst-in
    (implies (and (inv MT MA)
                  (DQ-stg-p (INST-stg i))
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA))
             (not (equal (MT-DQ-len MT) 0)))
  :hints (("Goal" :in-theory (enable DQ-stg-p))))
(in-theory (disable MT-DQ-len-non-zero-if-DQ-inst-in))

(encapsulate nil
(local
(defthm MT-last-DQ-index-before-if-car-is-IFU-if-DQ-len-zero-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (equal (MT-DQ-len MT) 0)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (last-DQ-index-before sub trace 0) 0))
  :hints (("Goal" :in-theory (enable MT-DQ-len-non-zero-if-DQ-inst-in)))))

(defthm MT-last-DQ-index-before-if-car-is-IFU-if-DQ-len-zero
    (implies (and (inv MT MA)
                  (consp trace)
                  (equal (MT-DQ-len MT) 0)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (MT-last-DQ-index-before trace MT) 0))
  :hints (("Goal" :in-theory (enable MT-last-DQ-index-before))))
)

(defun max-DQ-stg (MT) (list 'DQ (1- (MT-DQ-len MT))))
(in-theory (disable max-DQ-stg))

(defthm uniq-inst-at-max-DQ-stg
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (MT-DQ-len MT) 0)))
             (uniq-inst-at-stg (max-DQ-stg MT) MT))
  :hints (("Goal" :in-theory (enable max-DQ-stg))))

(defthm DQ-stg-idx-max-dq
    (equal (DQ-stg-idx (MAX-DQ-stg MT)) (1- (MT-DQ-len MT)))
  :hints (("Goal" :in-theory (enable max-DQ-stg dq-stg-idx))))

(defthm DQ-stg-p-max-DQ-stg
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (MT-DQ-len MT) 0)))
             (DQ-stg-p (max-DQ-stg MT)))
  :hints (("Goal" :in-theory (enable max-DQ-stg DQ-stg-p))))

(encapsulate nil
(local
(defthm hack-1
    (implies (equal stg (MAX-DQ-stg MT))
             (equal (DQ-stg-idx stg) (1- (MT-DQ-len MT))))))
```

```
(local
(defthm DQ-stg-idx-lt-MT-DQ-len
    (implies (and (inv MT MA)
                  (INST-in j MT) (INST-p j)
                  (MAETT-p MT) (MA-state-p MA)
                  (DQ-stg-p (INST-stg j)))
             (< (DQ-stg-idx (INST-stg j)) (MT-DQ-len MT)))
  :hints (("Goal" :in-theory (enable DQ-stg-p)))
  :rule-classes :linear))

; Cf.  DQ0-is-earlier-than-other-DQ
(local
(defthm no-DQ-inst-after-DQ-max-stg
    (implies (and (inv MT MA)
                  (INST-in j MT) (INST-p j)
                  (inst-in-order-p i j MT)
                  (equal (INST-stg i) (max-DQ-stg MT))
                  (MAETT-p MT) (MA-state-p MA))
             (not (DQ-stg-p (INST-stg j))))
  :hints (("Goal" :use ((:instance dq-stg-index-monotonic)
                        (:instance DQ-stg-p-max-DQ-stg))
                  :in-theory (disable DQ-stg-p-max-DQ-stg)))))

(local
(defthm last-dq-index-before-if-trace-is-after-max-DQ
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (subtrace-p trace MT) (INST-listp trace)
                  (subtrace-after-p i trace MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) (max-DQ-stg MT)))
             (equal (last-dq-index-before sub trace count) count))
  :hints (("Goal" :restrict ((no-DQ-inst-after-DQ-max-stg
                              ((i i)))))))

(local
 (defthm not-uniq-inst-at-max-dq-stg-if-car-is-IFU
     (implies (and (inv MT MA)
                   (subtrace-p trace MT) (INST-listp trace)
                   (dq-stg-p stg)
                   (consp trace) (IFU-stg-p (INST-stg (car trace)))
                   (MAETT-p MT) (MA-state-p MA))
              (not (uniq-inst-at-stg-in-trace stg trace)))
   :Hints (("Goal" :do-not-induct t
                   :cases ((consp (cdr trace)))
                   :expand (uniq-inst-at-stg-in-trace stg trace)))))

(local
(defthm MT-last-DQ-index-before-if-car-is-IFU-if-DQ-len-pos-help
    (implies (and (inv MT MA)
                  (consp sub)
                  (IFU-stg-p (INST-stg (car sub)))
                  (uniq-inst-at-stg-in-trace (max-DQ-stg MT) trace)
                  (tail-p sub trace)
                  (not (equal (MT-DQ-len MT) 0))
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (last-DQ-index-before sub trace count)
                    (MT-DQ-len MT)))
  :Hints (("Goal" :restrict ((last-dq-index-before-if-trace-is-after-max-DQ
                              ((i (car trace)))))))))
```

```
(defthm MT-last-DQ-index-before-if-car-is-IFU-if-DQ-len-pos
    (implies (and (inv MT MA)
                  (consp trace)
                  (IFU-stg-p (INST-stg (car trace)))
                  (not (equal (MT-DQ-len MT) 0))
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (MT-last-DQ-index-before trace MT)
                    (MT-DQ-len MT)))
  :hints (("Goal" :in-theory (e/d (MT-last-DQ-index-before
                                   uniq-inst-at-stg subtrace-p)
                                  (uniq-inst-at-max-DQ-stg))
                  :use (:instance uniq-inst-at-max-DQ-stg))))
)

(defthm MT-last-DQ-index-before-if-car-is-IFU
    (implies (and (inv MT MA)
                  (consp trace)
                  (IFU-stg-p (INST-stg (car trace)))
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (MT-last-DQ-index-before trace MT)
                    (MT-DQ-len MT)))
  :hints (("Goal" :cases ((equal (MT-DQ-len MT) 0)))))

(defthm MT-last-DQ-index-before-MT-trace
    (equal (MT-last-DQ-index-before (MT-trace MT) MT) 0)
  :hints (("Goal" :in-theory (enable MT-last-DQ-index-before))))

(encapsulate nil
(local
(defthm DQ-stg-idx-car-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (IN-order-DQ-trace-p trace idx)
                  (consp sub) (DQ-stg-p (INST-stg (car sub)))
                  (tail-p sub trace) (MAETT-p MT) (MA-state-p MA))
             (equal (last-DQ-index-before sub trace idx)
                    (DQ-stg-idx (INST-stg (car sub)))))
  :hints ((when-found (IFU-STG-P (INST-STG (CAR TRACE)))
                      (:cases ((consp (cdr trace))))))))

(defthm DQ-stg-idx-car
    (implies (and (inv MT MA)
                  (consp trace) (subtrace-p trace MT) (INST-listp trace)
                  (DQ-stg-p (INST-stg (car trace)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (DQ-stg-idx (INST-stg (car trace)))
                    (MT-last-DQ-index-before trace MT)))
  :hints (("Goal" :in-theory (enable MT-last-DQ-index-before
                                     subtrace-p
                                     inv In-order-DQ-p)
                  :restrict ((DQ-stg-idx-car-help
                              ((MT MT) (MA MA)))))))
)

(defthm MT-last-DQ-index-before-if-car-trace-at-DQ-idx
    (implies (and (inv MT MA)
                  (consp trace) (subtrace-p trace MT) (INST-listp trace)
                  (equal (INST-stg (car trace)) (list 'DQ idx))
                  (integerp idx) (<= 0 idx) (< idx 4)
```

```
                (MAETT-p MT) (MA-state-p MA))
           (equal (MT-last-DQ-index-before trace MT) idx))
  :hints (("Goal" :use (:instance DQ-stg-idx-car)
                  :in-theory (e/d (DQ-stg-p dq-stg-idx) (DQ-stg-idx-car)))))

(encapsulate nil
(local
(defthm in-order-dq-p-induction-1
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (b1p (dispatch-inst? MA))
                  (MAETT-p MT) (MA-state-p MA))
             (in-order-DQ-trace-p (step-trace trace MT MA sigs ISA spc smc)
                                  (coerce-DQ-stg
                                   (1- (MT-last-DQ-index-before trace MT)))))
  :hints (("Goal" :in-theory (enable MT-step in-order-DQ-p
                                     new-dq-stage
                                     DQ-stg-p))
          (when-found (INST-STG (STEP-INST (CAR TRACE) MT MA SIGS))
                      (:cases ((IFU-stg-p (INST-stg (car trace)))
                               (DQ-stg-p (INST-stg (car trace))))))
          (when-found (IFU-STG-P (INST-STG (CAR TRACE)))
                      (:cases ((consp (cdr trace))))))
  :rule-classes nil))

(local
(defthm in-order-dq-p-induction-2
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (not (b1p (dispatch-inst? MA)))
                  (MAETT-p MT) (MA-state-p MA))
             (in-order-DQ-trace-p (step-trace trace MT MA sigs ISA spc smc)
                                  (MT-last-DQ-index-before trace MT)))
  :hints (("Goal" :in-theory (enable MT-step in-order-DQ-p))
          (when-found (IFU-STG-P (INST-STG (STEP-INST (CAR TRACE) MT MA SIGS)))
                      (:cases ((IFU-stg-p (INST-stg (car trace)))
                               (DQ-stg-p (INST-stg (car trace))))))
          (when-found (IFU-STG-P (INST-STG (CAR TRACE)))
                      (:cases ((consp (cdr trace))))))
  :rule-classes nil))

(defthm in-order-dq-p-preserved
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA))
             (IN-order-DQ-p (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable MT-step in-order-DQ-p
                                     inv)
                  :use ((:instance in-order-dq-p-induction-1
                                   (trace (MT-trace MT))
                                   (ISA (MT-init-ISA MT))
                                   (spc 0) (smc 0))
                        (:instance in-order-dq-p-induction-2
                                   (trace (MT-trace MT))
                                   (ISA (MT-init-ISA MT))
                                   (spc 0) (smc 0))))))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Proof about IN-order-rob-p
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Proof of in-order-rob-p for initial states
(defthm in-order-rob-p-init-MT
```

```
    (in-order-rob-p (init-MT MA))
  :hints (("goal" :in-theory (enable init-MT in-order-rob-p))))

;;;; Proof of in-order-rob-p-preserved
(defun ROB-index-at (sub trace rix)
  (if (endp trace) rix
      (if (equal sub trace) rix
          (if (or (execute-stg-p (INST-stg (car trace)))
                  (complete-stg-p (INST-stg (car trace))))
              (rob-index-at sub (cdr trace) (rob-index (1+ rix)))
              (rob-index-at sub (cdr trace) rix)))))

(defun MT-ROB-index-at (sub MT)
  (ROB-index-at sub (MT-trace MT) (MT-ROB-head MT)))

(in-theory (disable MT-ROB-index-at))

(defthm MT-rob-index-at-MT-trace
    (equal (MT-rob-index-at (MT-trace MT) MT) (MT-rob-head MT))
  :hints (("Goal" :in-theory (enable MT-rob-index-at))))

(encapsulate nil
(local
 (defthm INST-rob-car-MT-rob-index-at-help-help
     (implies (and (inv MT MA)
                   (tail-p sub trace)
                   (subtrace-p trace MT) (INST-listp trace)
                   (consp trace)
                   (dispatched-p (car sub))
                   (consp sub)
                   (MAETT-p MT) (MA-state-p MA))
              (not (DQ-stg-p (INST-stg (car trace)))))
   :hints (("Goal" :cases ((equal sub trace))
                   :in-theory (e/d (dispatched-p)
                                   (inst-in-order-dispatched-undispatched)))
           ("subgoal 2" :use (:instance inst-in-order-dispatched-undispatched
                                        (i (car sub)) (j (car trace))))))))

(local
(defthm INST-rob-car-MT-rob-index-at-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (in-order-rob-trace-p trace idx)
                  (consp sub)
                  (tail-p sub trace)
                  (not (committed-p (car sub)))
                  (dispatched-p (car sub))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (equal (INST-tag (car sub)) (rob-index-at sub trace idx))
                    T))
  :hints (("Goal" :in-theory (enable committed-p dispatched-p)
                  :restrict ((INST-rob-car-MT-rob-index-at-help-help
                              ((sub sub)))))
          (when-found (IFU-STG-P (INST-STG (CAR TRACE)))
                      (:cases ((consp (cdr trace))))))))

(defthm INST-rob-car-MT-rob-index-at
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (consp trace)
                  (not (committed-p (car trace)))
                  (dispatched-p (car trace))
```

```
                (MAETT-p MT) (MA-state-p MA))
           (equal (equal (INST-tag (car trace)) (MT-rob-index-at trace MT))
                  T))
  :hints (("Goal" :in-theory (enable MT-rob-index-at subtrace-p
                                     inv in-order-rob-p)
                  :restrict ((INST-rob-car-MT-rob-index-at-help
                              ((MT MT) (MA MA)))))))

(encapsulate nil
(local
(defthm MT-rob-index-at-=-MT-rob-head-if-rob-empty-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (b1p (rob-empty? (MA-rob MA)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (rob-index-at sub trace idx) idx))
  :hints (("Goal" :in-theory (enable dispatched-p committed-p)))))

(defthm MT-rob-index-at-=-MT-rob-head-if-rob-empty
    (implies (and (inv MT MA)
                  (b1p (rob-empty? (MA-rob MA)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (equal (MT-rob-head MT) (MT-rob-index-at trace MT)) t))
  :hints (("Goal" :in-theory (enable MT-rob-index-at))))
)

(defun MT-last-robe (MT)
  (if (equal (MT-rob-tail MT) 0) 7 (1- (MT-rob-tail MT))))
(in-theory (disable MT-last-robe))

(defthm rob-index-p-MT-last-robe
    (implies (MAETT-p MT) (rob-index-p (MT-last-robe MT)))
  :hints (("goal" :in-theory (enable MT-last-robe))))

(defthm MT-last-robe-+-1
    (implies (MAETT-p MT)
             (equal (rob-index (1+ (MT-last-robe MT))) (MT-rob-tail MT)))
  :hints (("Goal" :in-theory (enable MT-last-robe rob-index))))

(defthm uniq-inst-at-mt-last-robe-if-not-rob-empty
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (rob-empty? (MA-rob MA)))))
             (uniq-inst-of-tag (MT-last-robe MT) MT))
  :hints (("Goal" :in-theory
                  (e/d (consistent-robe-p rob-empty?
                        MT-last-robe equal-b1p-converter lift-b-ops)
                       (CONSISTENT-ROBE-P-NTH-ROBE))
                  :use (:instance CONSISTENT-ROBE-P-NTH-ROBE
                                  (rix (MT-last-robe MT))))))

;; Proof of in-order-rob-p-preserved-if-flushed
(defthm not-MT-all-commit-before-execute-if-flush-all
    (implies (and (inv MT MA)
                  (consp trace)
                  (subtrace-p trace MT) (INST-listp trace)
                  (B1P (FLUSH-ALL? MA SIGS))
                  (EXECUTE-STG-P (INST-STG (CAR TRACE)))
                  (MAETT-p MT) (MA-state-p MA))
             (not (MT-ALL-COMMIT-BEFORE-TRACE TRACE MT)))
  :hints (("Goal" :in-theory (e/d (flush-all? lift-b-ops)
```

```
                                (not-all-commit-before-if-execute-stg-p))
                  :use (:instance not-all-commit-before-if-execute-stg-p
                                (i (car trace))))))

(defthm no-complete-stg-p-step-inst-if-inst-cause-jmp
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (complete-stg-p (INST-stg i))
                  (b1p (INST-cause-jmp? i MT MA sigs)))
             (not (complete-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("Goal" :in-theory
                  (disable committed-p-step-inst-if-INST-cause-jmp)
                  :use (:instance committed-p-step-inst-if-INST-cause-jmp))))

(defthm in-order-rob-p-preserved-if-flushed
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (in-order-ROB-trace-p trace rix)
                  (MT-all-commit-before-trace trace MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (flush-all? MA sigs)))
             (in-order-ROB-trace-p (step-trace trace MT MA sigs ISA spc smc)
                                   0))
  :hints (("Goal" :in-theory (enable committed-p* ))))

;;; Proof of in-order-rob-p-preserved-if-no-commit
(local
 (defthm in-order-rob-trace-p-mt-rob-head
     (implies (inv MT MA)
              (in-order-rob-trace-p (MT-trace MT) (MT-rob-head MT)))
   :hints (("Goal" :in-theory (enable inv in-order-rob-p)))))

(local
(defthm rob-index-1+-inst-tag
    (implies (inst-p i)
             (equal (rob-index (1+ (inst-tag i)))
                    (if (equal (inst-tag i) 7)
                        0 (1+ (inst-tag i)))))
  :hints (("Goal" :in-theory (enable rob-index-p unsigned-byte-p)))))
(local (in-theory (disable rob-index-1+-inst-tag)))

(local
 (defthm not-j-lt-i+1-if-i-lt-j
     (implies (and (inv MT MA)
                   (<= (INST-tag i) (INST-tag j))
                   (not (equal i j))
                   (INST-in i MT) (INST-p i)
                   (INST-in j MT) (INST-p j)
                   (MAETT-p MT) (MA-state-p MA)
                   (not (committed-p i)) (dispatched-p i)
                   (not (committed-p j)) (dispatched-p j))
              (not (< (INST-tag j) (rob-index (1+ (INST-tag i))))))
   :hints (("Goal" :in-theory (e/d (rob-index-1+-inst-tag)
                                   (tag-identity))
                   :use (:instance TAG-IDENTITY)))))

(encapsulate nil
(local
(defthm INST-in-order-inst-of-tag-if-not-rob-flg-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
```

```
                (MAETT-p MT) (MA-state-p MA)
                (in-order-rob-trace-p trace rix)
                (member-equal i trace) (INST-p i)
                (member-equal j trace) (INST-p j)
                (not (committed-p i)) (dispatched-p i)
                (not (committed-p j)) (dispatched-p j)
                (not (b1p (MT-rob-flg MT)))
                (<= rix (INST-tag i))
                (< (INST-tag i) (INST-tag j)))
           (member-in-order i j trace))
  :hints (("Goal" :in-theory (e/d ( committed-p dispatched-p
                                                member-in-order*)
                                  (INST-IS-AT-ONE-OF-THE-STAGES
                                   NOT-COMMITTED-P-IF-NOT-COMMIT-RETIRE))))
  :rule-classes nil))

(defthm INST-in-order-inst-of-tag-if-not-rob-flg
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (INST-in i MT) (INST-p i)
                (INST-in j MT) (INST-p j)
                (not (committed-p i)) (dispatched-p i)
                (not (committed-p j)) (dispatched-p j)
                (not (b1p (MT-rob-flg MT)))
                (< (INST-tag i) (INST-tag j)))
           (INST-in-order-p i j MT))
  :hints (("Goal" :use
                ((:instance INST-in-order-inst-of-tag-if-not-rob-flg-help
                            (trace (MT-trace MT)) (rix (MT-rob-head MT)))
                 (:instance CONSISTENT-ROBE-P-NTH-ROBE
                            (rix (INST-tag i))))
                :in-theory (e/d (inst-in-order-p INST-in in-order-rob-p
                                                 consistent-robe-p)
                                (consistent-robe-p-nth-robe)))))
)

(encapsulate nil
(local
 (defthm help-lemma1
     (implies (and (inv MT MA)
                   (<= (INST-tag j) (INST-tag i))
                   (INST-in i MT) (INST-p i)
                   (INST-in j MT) (INST-p j)
                   (not (equal j i))
                   (<= (MT-rob-tail MT) (INST-tag j))
                   (not (committed-p i)) (dispatched-p i)
                   (not (committed-p j)) (dispatched-p j)
                   (MAETT-p MT) (MA-state-p MA))
              (not (< (rob-index (+ 1 (INST-tag j)))
                      (MT-rob-tail MT))))
   :hints (("Goal" :in-theory (e/d (rob-index-1+-inst-tag)
                                   (tag-identity))
                   :use (:instance tag-identity)))))

(local
(defthm INST-in-order-inst-of-tag-if-rob-flg-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (in-order-rob-trace-p trace rix)
                (member-equal i trace)
                (member-equal j trace)
                (not (committed-p i)) (dispatched-p i)
```

```
                (not (committed-p j)) (dispatched-p j)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (MT-rob-flg MT))
                (<= rix (INST-tag i))
                (<= (MT-rob-tail MT) rix)
                (< (INST-tag j) (MT-rob-tail MT)))
           (member-in-order i j trace))
  :hints (("Goal" :in-theory (e/d ( committed-p dispatched-p
                                                member-in-order*)
                                  (INST-IS-AT-ONE-OF-THE-STAGES
                                   NOT-COMMITTED-P-IF-NOT-COMMIT-RETIRE))))
  :rule-classes nil))

(defthm INST-in-order-inst-of-tag-if-rob-flg
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (INST-in j MT) (INST-p j)
                (not (committed-p i)) (dispatched-p i)
                (not (committed-p j)) (dispatched-p j)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (MT-rob-flg MT))
                (<= (MT-rob-head MT) (INST-tag i))
                (< (INST-tag j) (MT-rob-tail MT)))
           (INST-in-order-p i j MT))
  :hints (("Goal" :use (:instance INST-in-order-inst-of-tag-if-rob-flg-help
                                  (trace (MT-trace MT)) (rix (MT-rob-head MT)))
                :in-theory (enable inst-in
                                   inst-in-order-p))))
)

(encapsulate nil
(local
(defthm INST-in-order-inst-of-tag-if-gt-rob-head-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (in-order-rob-trace-p trace rix)
                (MAETT-p MT) (MA-state-p MA)
                (not (committed-p i)) (dispatched-p i)
                (not (committed-p j)) (dispatched-p j)
                (member-equal i trace)
                (member-equal j trace)
                (<= rix (INST-tag i))
                (< (INST-tag i) (INST-tag j)))
           (member-in-order i j trace))
  :hints (("Goal" :in-theory (enable member-in-order* committed-p
                                  dispatched-p)))
  :rule-classes nil))

(defthm INST-in-order-inst-of-tag-if-gt-rob-head
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (not (committed-p i)) (dispatched-p i)
                (not (committed-p j)) (dispatched-p j)
                (INST-in i MT) (INST-p i)
                (INST-in j MT) (INST-p j)
                (<= (MT-rob-head MT) (INST-tag i))
                (< (INST-tag i) (INST-tag j)))
           (INST-in-order-p i j MT))
  :Hints (("Goal" :in-theory (enable inst-in INST-in-order-p)
                :use (:instance
                      INST-in-order-inst-of-tag-if-gt-rob-head-help
                      (trace (MT-trace MT)) (rix (MT-rob-head MT)))))))
```

```
)

(encapsulate nil
(local
(defthm rob-index-1+-INST-tag-=-0
    (implies (and (INST-p k) (<= (MT-rob-tail MT) (INST-tag k))
                  (< (rob-index (1+ (INST-tag k))) (MT-rob-tail MT)))
             (equal (rob-index (1+ (INST-tag k))) 0))
  :hints (("Goal" :in-theory (enable rob-index-1+-inst-tag)))))

(local
(defthm INST-in-order-inst-of-tag-if-le-rob-tail-induct-2
    (implies (and (inv MT MA)
                  (in-order-rob-trace-p trace rix)
                  (MAETT-p MT) (MA-state-p MA)
                  (<= rix (INST-tag i))
                  (subtrace-p trace MT) (INST-listp trace)
                  (member-equal i trace)
                  (member-equal j trace)
                  (not (committed-p i)) (dispatched-p i)
                  (not (committed-p j)) (dispatched-p j)
                  (< (INST-tag i) (INST-tag j))
                  (< (INST-tag j) (MT-rob-tail MT)))
             (member-in-order i j trace))
  :hints (("Goal" :in-theory (enable member-in-order* committed-p
                                     dispatched-p)))))

(local
(defthm INST-in-order-inst-of-tag-if-le-rob-tail-induct
    (implies (and (inv MT MA)
                  (in-order-rob-trace-p trace rix)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (member-equal i trace)
                  (member-equal j trace)
                  (not (committed-p i)) (dispatched-p i)
                  (not (committed-p j)) (dispatched-p j)
                  (<= (MT-rob-tail MT) rix)
                  (< (INST-tag i) (INST-tag j))
                  (< (INST-tag j) (MT-rob-tail MT)))
             (member-in-order i j trace))
  :hints (("Goal" :in-theory (enable member-in-order* committed-p
                                     dispatched-p)))
  :rule-classes nil))

(local
(defthm INST-in-order-inst-of-tag-if-le-rob-tail-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (not (committed-p i)) (dispatched-p i)
                  (not (committed-p j)) (dispatched-p j)
                  (b1p (MT-rob-flg MT))
                  (< (INST-tag i) (INST-tag j))
                  (< (INST-tag j) (MT-rob-tail MT)))
             (INST-in-order-p i j MT))
  :hints (("Goal" :use (:instance
                        INST-in-order-inst-of-tag-if-le-rob-tail-induct
                        (trace (MT-trace MT)) (rix (MT-rob-head MT)))
                  :in-theory
                  (e/d (INST-in INST-in-order-p) ())))))
```

```
(defthm INST-in-order-inst-of-tag-if-le-rob-tail
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (not (committed-p i)) (dispatched-p i)
                  (not (committed-p j)) (dispatched-p j)
                  (< (INST-tag i) (INST-tag j))
                  (< (INST-tag j) (MT-rob-tail MT)))
             (INST-in-order-p i j MT))
  :hints (("Goal" :cases ((b1p (MT-rob-flg MT))))))
)

; If ROB index of instruction i and j satisfy tag-in-order,
; instruction i precedes j in program order.
(defthm INST-in-order-inst-of-tag-if-tag-in-order
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (committed-p i)) (dispatched-p i)
                  (not (committed-p j)) (dispatched-p j)
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (tag-in-order (INST-tag i) (INST-tag j) MT))
             (INST-in-order-p i j MT))
  :hints (("Goal" :in-theory (enable tag-in-order))))

(defthm tag-in-order-MT-last-robe
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (rob-index-p rix)
                  (b1p (robe-valid? (nth-robe rix (MA-rob MA))))
                  (not (equal (MT-last-robe MT) rix)))
             (tag-in-order rix (MT-last-robe MT) MT))
  :hints (("Goal" :in-theory (e/d (tag-in-order MT-last-robe
                                                consistent-robe-p
                                                rob-index-p unsigned-byte-p)
                                  (CONSISTENT-ROBE-P-NTH-ROBE))
                  :use (:instance CONSISTENT-ROBE-P-NTH-ROBE))))

; The instruction stored in ROB at the entry designated by
; (MT-last-robe MT) is the last instruction in the ROB.
(defthm INST-in-order-inst-of-tag-MT-last-robe
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (rob-index-p rix)
                  (uniq-inst-of-tag rix MT)
                  (not (equal (MT-last-robe MT) rix))
                  (uniq-inst-of-tag (MT-last-robe MT) MT))
             (inst-in-order-p (inst-of-tag rix MT)
                              (inst-of-tag (MT-last-robe MT) MT)
                              MT))
  :hints (("Goal" :cases ((b1p (robe-valid? (nth-robe rix (MA-rob MA))))))))

(encapsulate nil
(local
(defthm not-undispatched-inst-after-mt-last-robe-help
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (MAETT-p MT) (MA-state-p MA)
                  (or (execute-stg-p (INST-stg i))
```

```
                    (complete-stg-p (INST-stg i)))
                (or (execute-stg-p (INST-stg j))
                    (complete-stg-p (INST-stg j)))
                (equal (INST-tag i) (MT-last-robe MT)))
           (not (inst-in-order-p i j MT)))
  :hints (("Goal" :in-theory (disable INST-in-order-inst-of-tag-MT-last-robe)
                  :use (:instance INST-in-order-inst-of-tag-MT-last-robe
                                  (rix (INST-tag j)))))))

(defthm not-undispatched-inst-after-mt-last-robe
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (MAETT-p MT) (MA-state-p MA)
                  (dispatched-p j)
                  (equal (INST-tag i) (MT-last-robe MT))
                  (or (execute-stg-p (INST-stg i))
                      (complete-stg-p (INST-stg i))))
             (not (inst-in-order-p i j MT)))
  :hints (("Goal" :use (:instance inst-is-at-one-of-the-stages (i j))
                  :in-theory (disable inst-is-at-one-of-the-stages)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (INST-in i MT) (INST-p i)
                           (INST-in j MT) (INST-p j)
                           (MAETT-p MT) (MA-state-p MA)
                           (inst-in-order-p i j MT)
                           (dispatched-p j)
                           (or (execute-stg-p (INST-stg i))
                               (complete-stg-p (INST-stg i))))
                      (not (equal (INST-tag i) (MT-last-robe MT)))))))
)

(defthm rob-index-at-subtrace-after-last-robe
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (subtrace-p trace MT) (INST-listp trace)
                  (subtrace-after-p i trace MT)
                  (equal (INST-tag i) (MT-last-robe MT))
                  (or (execute-stg-p (INST-stg i))
                      (complete-stg-p (INST-stg i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (rob-index-at sub trace idx) idx))
  :hints (("Goal" :restrict
                  (((:rewrite not-undispatched-inst-after-mt-last-robe . 2)
                    ((j (car trace))))))))

(encapsulate nil
(local
(defthm MT-rob-index-at-MT-rob-tail-if-no-rob-empty-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (tail-p sub trace)
                  (in-order-rob-trace-p trace idx)
                  (uniq-inst-of-tag-in-trace (MT-last-robe MT) trace)
                  (consp sub)
                  (DQ-stg-p (INST-stg (car sub)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (rob-index-at sub trace idx) (MT-rob-tail MT)))
  :Hints (("Goal" :restrict ((rob-index-at-subtrace-after-last-robe
```

```
                            ((i (car trace)))))
                  :in-theory (enable committed-p dispatched-p)))))

(defthm MT-rob-index-at-MT-rob-tail-if-no-rob-empty
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (consp trace)
                  (not (b1p (rob-empty? (MA-rob MA))))
                  (DQ-stg-p (INST-stg (car trace)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (equal (MT-rob-tail MT) (MT-rob-index-at trace MT)) t))
  :hints (("Goal" :in-theory (enable MT-rob-index-at uniq-inst-of-tag
                                     subtrace-p inv in-order-rob-p)
                  :use (:instance uniq-inst-at-mt-last-robe-if-not-rob-empty)
                  :restrict ((MT-rob-index-at-MT-rob-tail-if-no-rob-empty-help
                              ((MA MA) (MT MT)))))))
)

(defthm MT-rob-index-at-MT-rob-tail-if-DQ-stg-car
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (consp trace)
                  (DQ-stg-p (INST-stg (car trace)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (equal (MT-rob-tail MT) (MT-rob-index-at trace MT)) t))
  :hints (("Goal" :cases ((b1p (rob-empty? (MA-rob MA)))))))

(defthm INST-tag-step-inst-when-dispatched
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (DQ-stg-p (INST-stg i))
                  (or (execute-stg-p (INST-stg (step-INST i MT MA sigs)))
                      (complete-stg-p (INST-stg (step-INST i MT MA sigs))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-tag (step-INST i MT MA sigs))
                    (MT-rob-tail MT)))
  :hints (("Goal" :in-theory (enable step-inst-dq-inst step-inst-dq
                                     dispatch-inst))))

(encapsulate nil
(local
(defthm not-subtrace-after-if-undispatched-dispatched
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (subtrace-p trace MT) (INST-listp trace)
                  (consp trace) (dispatched-p (car trace))
                  (not (dispatched-p i))
                  (MAETT-p MT) (MA-state-p MA))
             (not (subtrace-after-p i trace MT)))
  :hints (("Goal" :use (:instance inst-in-order-dispatched-undispatched
                                  (i (car trace)) (j i))
                  :in-theory
                  (disable inst-in-order-dispatched-undispatched)))))

(defthm rob-index-at-subtrace-after-undispatched
    (implies (and (inv MT MA)
                  (not (dispatched-p i))
                  (subtrace-after-p i trace MT)
                  (INST-in i MT) (INST-p i)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (rob-index-at sub trace rix) rix))
```

```
  :hints (("Goal" :in-theory (enable dispatched-p))))
)

(encapsulate nil
(local
(defthm MT-rob-index-at-cdr-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (tail-p sub trace))
             (equal (rob-index-at (cdr sub) trace rix)
                    (if (and (dispatched-p (car sub))
                             (not (committed-p (car sub))))
                        (rob-index (1+ (rob-index-at sub trace rix)))
                        (rob-index-at sub trace rix))))
  :hints (("Goal" :in-theory (enable dispatched-p)))))

(defthm MT-rob-index-at-cdr
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (MT-rob-index-at (cdr trace) MT)
                    (if (and (dispatched-p (car trace))
                             (not (committed-p (car trace))))
                        (rob-index (1+ (MT-rob-index-at trace MT)))
                        (MT-rob-index-at trace MT))))
  :hints (("Goal" :in-theory (enable MT-rob-index-at subtrace-p))))

(defthm MT-rob-index-at-cdr-2
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (or (not (dispatched-p (car trace)))
                      (committed-p (car trace))))
             (equal (MT-rob-index-at trace MT)
                    (MT-rob-index-at (cdr trace) MT))))

(defthm rob-index-1+-MT-mt-rob-index-at
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (dispatched-p (car trace))
                  (not (committed-p (car trace))))
             (equal (rob-index (1+ (MT-rob-index-at trace MT)))
                    (MT-rob-index-at (cdr trace) MT))))
)
(in-theory (disable  rob-index-1+-MT-mt-rob-index-at
                     MT-rob-index-at-cdr-2))

(theory-invariant
 (not (and (member-equal '(:rewrite MT-rob-index-at-cdr) theory)
           (or (member-equal '(:rewrite rob-index-1+-MT-mt-rob-index-at)
                             theory)
               (member-equal '(:rewrite MT-rob-index-at-cdr-2) theory)))))

(defthm complete-stg-p-step-inst-if-not-commit-inst
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (complete-stg-p (INST-stg i))
                  (not (b1p (commit-inst? MA)))
                  (MAETT-p MT) (MA-state-p MA))
             (complete-stg-p (INST-stg (step-INST i MT MA sigs))))
  :hints (("Goal" :in-theory (enable step-inst-complete-inst
```

```
                                    step-inst-low-level-functions))))

(defthm not-execute-stg-step-inst-if-after-dq-stg
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (MAETT-p MT) (MA-state-p MA)
                  (DQ-stg-p (INST-stg i))
                  (INST-in-order-p i j MT))
             (not (execute-stg-p (INST-stg (step-inst j MT MA sigs)))))
  :hints (("Goal" :use ((:instance inst-is-at-one-of-the-stages (i j))
                        (:instance DQ-stg-index-monotonic))
                  :in-theory (e/d (step-inst-dq-inst
                                   step-inst-low-level-functions)
                                  (inst-is-at-one-of-the-stages)))))

(defthm not-complete-stg-step-inst-if-after-dq-stg
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (MAETT-p MT) (MA-state-p MA)
                  (DQ-stg-p (INST-stg i))
                  (INST-in-order-p i j MT))
             (not (complete-stg-p (INST-stg (step-inst j MT MA sigs)))))
  :hints (("Goal" :use ((:instance inst-is-at-one-of-the-stages (i j))
                        (:instance DQ-stg-index-monotonic))
                  :in-theory (e/d (step-inst-dq-inst
                                   step-inst-low-level-functions)
                                  (inst-is-at-one-of-the-stages)))))

(local
 (defthm in-order-rob-p-preserved-if-no-commit-help
     (implies (and (inv MT MA)
                   (DQ-stg-p (INST-stg i))
                   (subtrace-after-p i trace MT)
                   (INST-in i MT) (INST-p i)
                   (subtrace-p trace MT) (INST-listp trace)
                   (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
              (in-order-rob-trace-p (step-trace trace MT MA sigs
                                                ISA spc smc)
                                    count))
   :hints (("Goal" :in-theory (disable INST-IS-AT-ONE-OF-THE-STAGES)))))


(defthm in-order-rob-p-preserved-if-no-commit
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (commit-inst? MA))))
             (in-order-ROB-trace-p (step-trace trace MT MA sigs ISA spc smc)
                                   (MT-rob-index-at trace MT)))
  :hints (("goal" :in-theory (enable DISPATCHED-P
                                     committed-p))
          (when-found (COMPLETE-STG-P (INST-STG (STEP-INST (CAR TRACE)
                                                           MT MA SIGS)))
                      (:cases ((DQ-stg-p (INST-stg (car trace))))))
          (when-found (execute-STG-P (INST-STG (STEP-INST (CAR TRACE)
                                                          MT MA SIGS)))
                      (:cases ((DQ-stg-p (INST-stg (car trace)))))))
  :rule-classes nil)
```

```
;;; Proof of in-order-rob-p-preserved-if-commit-inst
(encapsulate nil
(local
(defthm mt-all-commit-before-cdr-if-car-is-not-commit-help
    (implies (and (consp sub) (tail-p sub trace)(INST-listp sub)
                  (not (committed-p (car sub))))
             (not (trace-all-commit-before-trace (cdr sub) trace)))))

(defthm mt-all-commit-before-cdr-if-car-is-not-commit
    (implies (and (consp trace) (subtrace-p trace MT) (INST-listp trace)
                  (not (committed-p (car trace))))
             (not (MT-all-commit-before-trace (cdr trace) MT)))
  :hints (("Goal" :in-theory (enable subtrace-p MT-all-commit-before-trace))))
)

(defthm not-MT-all-commit-before-DQ-if-commit-inst
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (DQ-stg-p (INST-stg i))
                  (b1p (commit-inst? MA))
                  (MAETT-p MT) (MA-state-p MA))
             (not (MT-all-commit-before i MT)))
  :hints (("goal" :cases ((uniq-inst-of-tag (MT-rob-head MT) MT)))
          ("subgoal 1" :use (:instance
                             no-commit-inst-precedes-if-all-commit-before
                             (i i) (j (inst-of-tag (MT-rob-head MT) MT)))
                       :in-theory
                       (disable no-commit-inst-precedes-if-all-commit-before))))

(defthm not-MT-all-commit-before-execute-if-commit-inst
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (execute-stg-p (INST-stg i))
                  (b1p (commit-inst? MA))
                  (MAETT-p MT) (MA-state-p MA))
             (not (MT-all-commit-before i MT)))
  :hints (("goal" :cases ((uniq-inst-of-tag (MT-rob-head MT) MT)))
          ("subgoal 1" :use
                       ((:instance
                         no-commit-inst-precedes-if-all-commit-before
                         (i i) (j (inst-of-tag (MT-rob-head MT) MT)))
                        (:instance UNCOMMITTED-INST-P-IS-AFTER-MT-ROB-HEAD))
                       :in-theory
                       (disable no-commit-inst-precedes-if-all-commit-before
                                UNCOMMITTED-INST-P-IS-AFTER-MT-ROB-HEAD))))

(defthm not-execute-stg-step-inst-if-MT-all-commit-before
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (b1p (commit-inst? MA))
                  (MT-all-commit-before i MT)
                  (MAETT-p MT) (MA-state-p MA))
             (not (execute-stg-p (INST-stg (step-inst i MT MA sigs)))))
  :hints (("Goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (disable inst-is-at-one-of-the-stages))))

(defthm not-complete-stg-step-inst-if-MT-all-commit-before
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (b1p (commit-inst? MA))
                  (MT-all-commit-before i MT)
                  (MAETT-p MT) (MA-state-p MA))
```

```
            (not (complete-stg-p (INST-stg (step-inst i MT MA sigs)))))
  :hints (("Goal" :use
             ((:instance inst-is-at-one-of-the-stages)
              (:instance UNCOMMITTED-INST-P-IS-AFTER-MT-ROB-HEAD)
              (:instance RETIRE-INST-STG-STEP-INST-IF-COMPLETE-ROBE-IS-HEAD))
             :in-theory
             (e/d (committed-p)
                  (RETIRE-INST-STG-STEP-INST-IF-COMPLETE-ROBE-IS-HEAD
                   inst-is-at-one-of-the-stages)))))

(defthm MT-all-commit-before-inst-commit
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (b1p (INST-commit? i MA))
                  (MAETT-p MT) (MA-state-p MA))
             (MT-all-commit-before i MT))
  :hints (("Goal" :in-theory (enable INST-commit? lift-b-ops))))

(defthm complete-stg-p-step-inst-if-not-MT-all-commit-before
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (MT-all-commit-before i MT))
                  (complete-stg-p (INST-stg i))
                  (MAETT-p MT) (MA-state-p MA))
             (complete-stg-p (INST-stg (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (enable step-inst-complete-inst
                                     lift-b-ops
                                     step-inst-low-level-functions))))

(encapsulate nil
(local
 (defthm in-order-rob-trace-p-cons-IFU
     (implies (IFU-stg-p (INST-stg i))
              (in-order-rob-trace-p (cons (step-inst i MT MA sigs)
                                          (step-trace trace mt ma sigs
                                                      isa spc smc))
                                    count))
   :hints (("Goal" :expand
                   (in-order-rob-trace-p (cons (step-inst i MT MA sigs)
                                               (step-trace trace mt ma sigs
                                                           isa spc smc))
                                         count)))))

(local
(defthm in-order-rob-p-preserved-induction3-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (equal rix (MT-rob-index-at trace MT))
                  (in-order-rob-trace-p trace (MT-rob-index-at trace MT))
                  (not (MT-all-commit-before-trace trace MT))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (commit-inst? MA)))
             (in-order-ROB-trace-p (step-trace trace MT MA sigs ISA spc smc)
                                   (MT-rob-index-at trace MT)))
  :hints (("goal" :in-theory (e/d (DISPATCHED-P committed-p)
                                  (ROBE-AT-HEAD-IF-MT-ALL-COMMIT-BEFORE)))
          (when-found (COMPLETE-STG-P (INST-STG (STEP-INST (CAR TRACE)
                                                           MT MA SIGS)))
                      (:cases ((DQ-stg-p (INST-stg (car trace))))))
          (when-found (execute-STG-P (INST-STG (STEP-INST (CAR TRACE)
                                                          MT MA SIGS)))
```

```
                    (:cases ((DQ-stg-p (INST-stg (car trace)))))))))

(defthm in-order-rob-p-preserved-if-commit-inst
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (in-order-rob-trace-p trace (MT-rob-index-at trace MT))
                  (MT-all-commit-before-trace trace MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (commit-inst? MA)))
     (in-order-ROB-trace-p (step-trace trace MT MA sigs ISA spc smc)
                           (rob-index (+ 1 (MT-rob-index-at trace MT)))))
  :hints (("Goal" :in-theory (e/d (COMMITTED-P dispatched-p
                                               MT-rob-index-at-cdr-2
                                               ROB-INDEX-1+-MT-MT-ROB-INDEX-AT)
                                  (IN-ORDER-ROB-TRACE-P-MT-ROB-HEAD
                                   MT-rob-index-at-cdr)))
          (when-found (FETCH-INST? MA SIGS)
                      (:in-theory (e/d (committed-p dispatched-p) ())))
          (when-found-multiple ((MT-ROB-INDEX-AT (CDR TRACE) MT)
                                (MT-ROB-INDEX-AT TRACE MT))
                               (:cases ((committed-p (car trace))
                                        (not (dispatched-p (car trace))))))
          (when-found-multiple ((EXECUTE-STG-P (INST-STG (CAR TRACE)))
                                (COMPLETE-STG-P (INST-STG (CAR TRACE)))
                                (COMMIT-STG-P (INST-STG (CAR TRACE)))
                                (RETIRE-STG-P (INST-STG (CAR TRACE))))
                               (:cases ((dq-stg-p (INST-stg (car trace)))
                                        (IFU-stg-p (INST-stg (car trace)))))))
  :rule-classes nil)
)

(defthm in-order-rob-p-preserved
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (in-order-ROB-p (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (e/d (in-order-rob-p MT-step
                                                  step-MT-rob-head
                                                  inv)
                                  (in-order-rob-trace-p step-trace))
                  :use ((:instance in-order-rob-p-preserved-if-no-commit
                                   (trace (MT-trace MT))
                                   (ISA (MT-init-ISA MT))
                                   (spc 0) (smc 0))
                        (:instance in-order-rob-p-preserved-if-commit-inst
                                   (trace (MT-trace MT))
                                   (ISA (MT-init-ISA MT))
                                   (spc 0) (smc 0))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;; Proof about in-order-LSU-inst-p
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Proof of in-order-LSU-inst-p for initial states.
(defthm in-order-LSU-inst-p-init-MT
    (in-order-LSU-inst-p (init-MT MA) MA)
  :hints (("goal" :in-theory (enable init-MT in-order-LSU-inst-p
                                     IN-ORDER-LOAD-STORE-P))))

;;;; Invariant proof
(defthm wbuf-stg-p*
    (equal (wbuf-stg-p stg)
```

```
          (or (wbuf0-stg-p stg) (wbuf1-stg-p stg)))
  :hints (("goal" :in-theory (enable wbuf-stg-p wbuf0-stg-p wbuf1-stg-p)))
  :rule-classes :definition)


(defthm not-wbuf0-stg-p-new-dq-stage
    (not (wbuf0-stg-p (new-DQ-stage MT MA)))
  :hints (("Goal" :in-theory (enable new-DQ-stage wbuf0-stg-p))))

(defthm not-wbuf1-stg-p-new-dq-stage
    (not (wbuf1-stg-p (new-dq-stage mt ma)))
  :hints (("Goal" :in-theory (enable new-DQ-stage wbuf1-stg-p))))

(defthm not-wbuf0-stg-p-step-inst-if-not-LSU-RS
    (implies (and (INST-p i)
                  (not (equal (INST-stg i) '(LSU RS0)))
                  (not (equal (INST-stg i) '(LSU RS1)))
                  (not (wbuf0-stg-p (INST-stg i)))
                  (not (wbuf1-stg-p (INST-stg i))))
             (not (wbuf0-stg-p (INST-stg (step-inst i MT MA sigs)))))
  :hints (("Goal" :in-theory (enable wbuf0-stg-p wbuf1-stg-p
                                     complete-stg-p
                                     commit-stg-p execute-stg-p
                                     BU-stg-p MU-stg-p LSU-stg-p
                                     step-inst-execute-inst
                                     step-inst-low-level-functions
                                     IU-stg-p inst-stg-step-inst)
                  :use ((:instance inst-is-at-one-of-the-stages)))))

(defthm not-wbuf0-stg-p-step-inst-if-not-execute-stg
    (implies (and (INST-p i)
                  (not (execute-stg-p (INST-stg i)))
                  (not (wbuf0-stg-p (INST-stg i)))
                  (not (wbuf1-stg-p (INST-stg i))))
             (not (wbuf0-stg-p (INST-stg (step-inst i MT MA sigs)))))
  :hints (("Goal" :in-theory (enable execute-stg-p LSU-stg-p))))

(defthm not-wbuf1-stg-p-step-inst-if-not-LSU-RS
    (implies (and (INST-p i)
                  (not (equal (INST-stg i) '(LSU RS0)))
                  (not (equal (INST-stg i) '(LSU RS1)))
                  (not (wbuf1-stg-p (INST-stg i))))
             (not (wbuf1-stg-p (INST-stg (step-inst i MT MA sigs)))))
  :hints (("Goal" :in-theory (enable wbuf1-stg-p
                                     complete-stg-p
                                     commit-stg-p execute-stg-p
                                     BU-stg-p MU-stg-p LSU-stg-p
                                     step-inst-execute-inst
                                     step-inst-low-level-functions
                                     IU-stg-p inst-stg-step-inst)
                  :use ((:instance inst-is-at-one-of-the-stages)))))

(defthm not-wbuf1-stg-p-step-inst-if-not-execute-stg
    (implies (and (INST-p i)
                  (not (execute-stg-p (INST-stg i)))
                  (not (wbuf1-stg-p (INST-stg i))))
             (not (wbuf1-stg-p (INST-stg (step-inst i MT MA sigs)))))
  :hints (("Goal" :in-theory (enable execute-stg-p LSU-stg-p))))

(local
(defthm not-simultaneous-issue-to-wbuf0-wbuf1
    (implies (and (inv MT MA)
```

```
                (wbuf0-stg-p (INST-stg (step-INST i MT MA sigs)))
                (INST-in i MT) (INST-p i)
                (INST-in j MT) (INST-p j)
                (or (equal (INST-stg i) '(LSU RS0))
                    (equal (INST-stg i) '(LSU RS1)))
                (or (equal (INST-stg j) '(LSU RS0))
                    (equal (INST-stg j) '(LSU RS1))))
           (not (wbuf1-stg-p (INST-stg (step-INST j MT MA sigs)))))
  :hints (("Goal" :in-theory (enable step-inst-execute-inst
                                     step-inst-low-level-functions)))))

(defthm not-both-inst-at-wbuf-after-step-inst
    (implies (and (inv MT MA)
                (wbuf0-stg-p (INST-stg (step-INST i MT MA sigs)))
                (INST-in i MT) (INST-p i)
                (INST-in j MT) (INST-p j)
                (MAETT-p MT) (MA-state-p MA)
                (not (wbuf-stg-p (INST-stg i)))
                (not (wbuf-stg-p (INST-stg j))))
           (not (wbuf1-stg-p (INST-stg (step-INST j MT MA sigs)))))
  :hints (("goal" :in-theory (e/d (execute-stg-p BU-stg-p
                                                 IU-stg-p MU-stg-p LSU-stg-p )
                                  (inst-is-at-one-of-the-stages))
                :use ((:instance inst-is-at-one-of-the-stages)
                      (:instance inst-is-at-one-of-the-stages (i j))))))

(defthm not-wbuf1-stg-step-inst-if-wbuf0-stg
    (implies (and (INST-p i) (wbuf0-stg-p (INST-stg i)))
           (not (wbuf1-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("Goal" :in-theory (enable wbuf0-stg-p))))

(defthm inst-in-order-wbuf0-wbuf1-step-inst-help1
    (implies (and (inv MT MA)
                (wbuf1-stg-p (INST-stg j))
                (INST-in i MT) (INST-p i)
                (INST-in j MT) (INST-p j)
                (MAETT-p MT) (MA-state-p MA)
                (not (wbuf-stg-p (INST-stg i))))
           (not (wbuf0-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("Goal" :in-theory (e/d (execute-stg-p LSU-stg-p BU-stg-p
                                   INST-SELECT-WBUF0? lift-b-ops
                                   inst-stg-step-inst MU-stg-p IU-stg-p)
                                  (inst-is-at-one-of-the-stages))
                :use (:instance inst-is-at-one-of-the-stages))))

(defthm inst-in-order-wbuf0-wbuf1-step-inst-help2
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (INST-in j MT) (INST-p j)
                (wbuf-stg-p (INST-stg i))
                (not (wbuf-stg-p (INST-stg j)))
                (wbuf1-stg-p (INST-stg (step-inst j MT MA sigs))))
           (INST-in-order-p i j MT))
  :hints (("Goal" :in-theory (e/d (wbuf-stg-p* wbuf0-stg-p wbuf1-stg-p
                                               INST-IN-ORDER-P-LSU-ISSUED-RS)
                                  (not-wbuf1-stg-p-step-inst-if-not-LSU-RS))
                :use (:instance not-wbuf1-stg-p-step-inst-if-not-LSU-RS
                                (i j))))))

(defthm inst-in-order-wbuf0-wbuf1-step-inst-help3
    (implies (and (inv MT MA)
                (wbuf0-stg-p (INST-stg (step-INST i MT MA sigs)))
```

```
                (INST-in i MT) (INST-p i)
                (INST-in j MT) (INST-p j)
                (wbuf1-stg-p (INST-stg i))
                (wbuf1-stg-p (INST-stg j)))
           (not (wbuf1-stg-p (INST-stg (step-INST j MT MA sigs)))))
  :hints (("Goal" :in-theory (enable inst-stg-step-inst lift-b-ops
                                     wbuf1-stg-p wbuf0-stg-p
                                     RELEASE-WBUF0?))))

(defthm inst-in-order-wbuf0-wbuf1-step-inst
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (INST-in j MT) (INST-p j)
                (MAETT-p MT) (MA-state-p MA)
                (wbuf0-stg-p (INST-stg (step-INST i MT MA sigs)))
                (wbuf1-stg-p (INST-stg (step-INST j MT MA sigs))))
           (INST-in-order-p i j MT))
  :hints (("Goal" :cases ((wbuf0-stg-p (INST-stg i))
                          (wbuf1-stg-p (INST-stg i)))
                :in-theory (enable wbuf-stg-p* INST-IN-ORDER-P-WBUF0-WBUF1)
                :restrict ((inst-in-order-wbuf0-wbuf1-step-inst-help1
                             ((j j)))))
          (use-hint-always (:cases ((wbuf0-stg-p (INST-stg j))
                                    (wbuf1-stg-p (INST-stg j)))))))
  :rule-classes
  ((:rewrite :corollary
     (implies (and (inv MT MA)
                   (wbuf1-stg-p (INST-stg (step-INST j MT MA sigs)))
                   (INST-in i MT) (INST-p i)
                   (INST-in j MT) (INST-p j)
                   (MAETT-p MT) (MA-state-p MA)
                   (INST-in-order-p j i MT))
              (not (wbuf0-stg-p (INST-stg (step-INST i MT MA sigs))))))))

(encapsulate nil
(local
(defthm in-order-wb-trace-p-mt-trace-MT-step-help
    (implies (and (inv MT MA)
                (subtrace-after-p i trace MT)
                (INST-in i MT) (INST-p i)
                (subtrace-p trace MT) (INST-listp trace)
                (wbuf1-stg-p (INST-stg (step-inst i MT MA sigs)))
                (MAETT-p MT) (MA-state-p MA))
           (no-inst-at-wbuf0-p (step-trace trace MT MA sigs ISA spc smc)))))

(defthm in-order-wb-trace-p-mt-trace-MT-step
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA))
           (in-order-wb-trace-p (step-trace trace MT MA sigs ISA spc smc)))
  :hints (("Goal" :restrict ((in-order-wb-trace-p-mt-trace-MT-step-help
                               ((i (car trace))))))))
)

(defthm not-simultaneous-issue-to-wbuf0-rbuf
    (implies (and (inv MT MA)
                (equal (INST-stg (step-INST i MT MA sigs))
                       '(LSU rbuf))
                (INST-in i MT) (INST-p i)
                (INST-in j MT) (INST-p j)
                (or (equal (INST-stg i) '(LSU RS0))
                    (equal (INST-stg i) '(LSU RS1)))
```

```
                (or (equal (INST-stg j) '(LSU RS0))
                    (equal (INST-stg j) '(LSU RS1))))
            (not (wbuf0-stg-p (INST-stg (step-INST j MT MA sigs)))))
  :hints (("Goal" :in-theory (enable step-inst-execute-inst lift-b-ops
                                     step-inst-low-level-functions
                                     LSU-RS0-ISSUE-READY?
                                     LSU-RS1-ISSUE-READY?
                                     ISSUE-LSU-RS0? ISSUE-LSU-RS1?))))

(defthm not-simultaneous-issue-to-wbuf1-rbuf
    (implies (and (inv MT MA)
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU rbuf))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (or (equal (INST-stg i) '(LSU RS0))
                      (equal (INST-stg i) '(LSU RS1)))
                  (or (equal (INST-stg j) '(LSU RS0))
                      (equal (INST-stg j) '(LSU RS1))))
             (not (wbuf1-stg-p (INST-stg (step-INST j MT MA sigs)))))
  :hints (("Goal" :in-theory (enable step-inst-execute-inst lift-b-ops
                                     step-inst-low-level-functions
                                     LSU-RS0-ISSUE-READY?
                                     LSU-RS1-ISSUE-READY?
                                     ISSUE-LSU-RS0? ISSUE-LSU-RS1?))))

(defthm rbuf-wbuf0-step-LSU-off-if-wbuf0-issued
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU rbuf))
                  (equal (INST-stg i) '(LSU rbuf))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (or (equal (INST-stg j) '(LSU RS0))
                      (equal (INST-stg j) '(LSU RS1)))
                  (b1p (rbuf-wbuf0? (LSU-rbuf (step-LSU MA sigs)))))
             (not (wbuf0-stg-p (INST-stg (step-INST j MT MA sigs)))))
  :hints (("Goal" :in-theory (enable step-inst-execute-inst
                                     step-LSU step-rbuf lift-b-ops
                                     RELEASE-RBUF?
                                     ISSUE-LSU-RS1? ISSUE-LSU-RS0?
                                     LSU-RS0-ISSUE-READY?
                                     LSU-RS1-ISSUE-READY?
                                     INST-SELECT-WBUF0?
                                     RELEASE-WBUF0?
                                     step-inst-low-level-functions))))

(defthm INST-in-order-wbuf0-rbuf-step-INST-help1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (not (wbuf-stg-p (INST-stg j)))
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU rbuf))
                  (wbuf0-stg-p (INST-stg (step-INST j MT MA sigs)))
                  (b1p (rbuf-wbuf0? (LSU-rbuf (step-LSU MA sigs)))))
             (INST-in-order-p j i MT))
  :hints (("Goal" :use ((:instance stages-reachable-to-LSU-rbuf)
                        (:instance not-wbuf0-stg-p-step-inst-if-not-LSU-RS
                                   (i j)))
```

```
                :in-theory (e/d (INST-IN-ORDER-P-LSU-ISSUED-RS)
                                (not-wbuf0-stg-p-step-inst-if-not-LSU-RS)))))

(defthm LSU-rbuf-wbuf0-step-LSU
    (implies (and (b1p (rbuf-valid? (LSU-rbuf (MA-LSU MA))))
                  (not (b1p (release-rbuf? (MA-LSU MA) MA sigs))))
             (equal (rbuf-wbuf0? (LSU-rbuf (step-LSU MA sigs)))
                    (rbuf-wbuf0? (LSU-rbuf (MA-LSU MA)))))
  :hints (("Goal" :in-theory (enable step-LSU step-rbuf lift-b-ops))))

(defthm INST-in-order-wbuf0-rbuf-step-INST-help2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (wbuf0-stg-p (INST-stg j))
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU rbuf))
                  (wbuf0-stg-p (INST-stg (step-INST j MT MA sigs)))
                  (b1p (rbuf-wbuf0? (LSU-rbuf (step-LSU MA sigs)))))
             (INST-in-order-p j i MT))
  :hints (("Goal" :use ((:instance stages-reachable-to-LSU-rbuf))
                  :restrict ((LSU-RBUF-VALID-IF-INST-IN ((i i)))
                             (INST-IN-ORDER-P-WBUF0-RBUF ((i j) (j i))))
                  :in-theory (e/d (INST-IN-ORDER-P-LSU-ISSUED-RS
                                   INST-IN-ORDER-P-WBUF0-RBUF
                                   INST-stg-step-inst
                                   wbuf0-stg-p lift-b-ops)
                                  ()))))

(defthm INST-in-order-wbuf0-rbuf-step-INST-help3
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (wbuf1-stg-p (INST-stg j))
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU rbuf))
                  (wbuf0-stg-p (INST-stg (step-INST j MT MA sigs)))
                  (b1p (rbuf-wbuf0? (LSU-rbuf (step-LSU MA sigs)))))
             (INST-in-order-p j i MT))
  :hints (("Goal" :use ((:instance stages-reachable-to-LSU-rbuf))
                  :restrict ((LSU-RBUF-VALID-IF-INST-IN ((i i)))
                             (INST-IN-ORDER-P-WBUF1-RBUF ((i j) (j i))))
                  :in-theory (e/d (INST-IN-ORDER-P-LSU-ISSUED-RS
                                   INST-IN-ORDER-P-WBUF0-RBUF
                                   INST-IN-ORDER-P-WBUF1-RBUF
                                   RELEASE-WBUF0?
                                   INST-stg-step-inst wbuf1-stg-p
                                   wbuf0-stg-p lift-b-ops)
                                  ()))))

(defthm INST-in-order-wbuf0-rbuf-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU rbuf))
                  (wbuf0-stg-p (INST-stg (step-INST j MT MA sigs)))
                  (b1p (rbuf-wbuf0? (LSU-rbuf (step-LSU MA sigs)))))
             (INST-in-order-p j i MT))
```

```
  :hints (("Goal" :cases ((wbuf0-stg-p (INST-stg j))
                          (wbuf1-stg-p (INST-stg j)))))
  :rule-classes
  ((:rewrite :corollary
     (implies (and (inv MT MA)
                   (equal (INST-stg (step-INST i MT MA sigs))
                          '(LSU rbuf))
                   (INST-in i MT) (INST-p i)
                   (INST-in j MT) (INST-p j)
                   (b1p (rbuf-wbuf0? (LSU-rbuf (step-LSU MA sigs))))
                   (INST-in-order-p i j MT)
                   (MAETT-p MT) (MA-state-p MA))
              (not (wbuf0-stg-p (INST-stg (step-INST j MT MA sigs))))))))

(encapsulate nil
(local
(defthm in-order-wbuf0-rbuf-step-trace-help
    (implies (and (inv MT MA)
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU rbuf))
                  (INST-in i MT) (INST-p i)
                  (subtrace-p trace MT) (INST-listp trace)
                  (subtrace-after-p i trace MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (rbuf-valid? (LSU-rbuf (step-LSU MA sigs))))
                  (b1p (rbuf-wbuf0? (LSU-rbuf (step-LSU MA sigs))))
                  (b1p (wbuf-valid? (LSU-wbuf0 (step-LSU MA sigs)))))
             (no-inst-at-wbuf0-p (step-trace trace MT MA sigs ISA spc smc)))))

(defthm in-order-wbuf0-rbuf-step-trace
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (rbuf-valid? (LSU-rbuf (step-LSU MA sigs))))
                  (b1p (rbuf-wbuf0? (LSU-rbuf (step-LSU MA sigs))))
                  (b1p (wbuf-valid? (LSU-wbuf0 (step-LSU MA sigs)))))
     (in-order-wbuf0-rbuf-p (step-trace trace MT MA sigs ISA spc smc))))
)

(defthm in-order-rbuf-wbuf0-step-inst-help1
    (implies (and (inv MT MA)
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU rbuf))
                  (wbuf0-stg-p (INST-stg (step-INST j MT MA sigs)))
                  (not (wbuf-stg-p (INST-stg j)))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (not (b1p (rbuf-wbuf0? (LSU-rbuf (step-LSU MA sigs))))))
             (INST-in-order-p i j MT))
  :hints (("Goal" :use ((:instance stages-reachable-to-LSU-rbuf)
                        (:instance not-wbuf0-stg-p-step-inst-if-not-LSU-RS
                                   (i j)))
                  :in-theory (e/d (INST-IN-ORDER-P-LSU-ISSUED-RS)
                                  (not-wbuf0-stg-p-step-inst-if-not-LSU-RS)))))

(defthm rbuf-wbuf0-step-LSU-unchanged-if-load-not-released
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU rbuf))
                  (wbuf0-stg-p (INST-stg (step-INST j MT MA sigs)))
                  (wbuf0-stg-p (INST-stg j))
```

```
                (INST-in i MT) (INST-p i)
                (INST-in j MT) (INST-p j)
                (equal (INST-stg i) '(LSU rbuf)))
           (equal (rbuf-wbuf0? (LSU-rbuf (step-LSU MA sigs)))
                  (rbuf-wbuf0? (LSU-rbuf (MA-LSU MA)))))
  :hints (("Goal" :in-theory (enable step-LSU step-rbuf lift-b-ops
                                     inst-stg-step-inst wbuf0-stg-p)
                :restrict ((LSU-RBUF-VALID-IF-INST-IN ((i i)))))))

(defthm rbuf-wbuf0-step-LSU-on-if-wbuf0-not-released
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU rbuf))
                  (wbuf0-stg-p (INST-stg (step-INST j MT MA sigs)))
                  (wbuf0-stg-p (INST-stg j))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (or (equal (INST-stg i) '(LSU RS0))
                      (equal (INST-stg i) '(LSU RS1))))
             (equal (rbuf-wbuf0? (LSU-rbuf (step-LSU MA sigs))) 1))
  :hints (("Goal" :in-theory (enable step-LSU step-rbuf lift-b-ops
                                     ISSUE-LSU-RS0? ISSUE-LSU-RS1?
                                     RELEASE-WBUF0? RELEASE-WBUF0-READY?
                                     inst-stg-step-inst wbuf0-stg-p))))

(defthm in-order-rbuf-wbuf0-step-inst-help2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU rbuf))
                  (wbuf0-stg-p (INST-stg (step-INST j MT MA sigs)))
                  (wbuf0-stg-p (INST-stg j))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (not (b1p (rbuf-wbuf0? (LSU-rbuf (step-LSU MA sigs))))))
             (INST-in-order-p i j MT))
  :hints (("Goal" :use ((:instance stages-reachable-to-LSU-rbuf))
                  :restrict ((LSU-RBUF-VALID-IF-INST-IN ((i i)))
                             (INST-IN-ORDER-P-LSU-ISSUED-RS ((i j) (j i)))
                             (INST-IN-ORDER-P-RBUF-WBUF0 ((i i) (j j))))
                  :in-theory (e/d (INST-IN-ORDER-P-LSU-ISSUED-RS
                                   INST-IN-ORDER-P-RBUF-WBUF0)
                                  ()))))

(defthm wbuf1-not-advance-if-rbuf-inst-in
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (wbuf1-stg-p (INST-stg j))
                  (equal (INST-stg i) '(LSU rbuf))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j))
             (not (wbuf0-stg-p (INST-stg (step-INST j MT MA sigs)))))
  :hints (("Goal" :in-theory (enable inst-stg-step-inst
                                     wbuf1-stg-p RELEASE-WBUF0?
                                     RELEASE-WBUF0-READY?
                                     lift-b-ops)
                  :restrict ((LSU-RBUF-VALID-IF-INST-IN ((i i)))))))

(defthm rbuf-wbuf0-step-LSU-on-if-wbuf1-advance
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
```

```
                (equal (INST-stg (step-INST i MT MA sigs))
                       '(LSU rbuf))
                (wbuf0-stg-p (INST-stg (step-INST j MT MA sigs)))
                (wbuf1-stg-p (INST-stg j))
                (INST-in i MT) (INST-p i)
                (INST-in j MT) (INST-p j)
                (or (equal (INST-stg i) '(LSU RS0))
                    (equal (INST-stg i) '(LSU RS1))))
           (equal (rbuf-wbuf0? (LSU-rbuf (step-LSU MA sigs))) 1))
  :hints (("Goal" :in-theory (enable step-LSU step-rbuf lift-b-ops
                                     wbuf1-stg-p RELEASE-WBUF0?
                                     inst-stg-step-inst))))

(defthm in-order-rbuf-wbuf0-step-inst-help3
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (equal (INST-stg (step-INST i MT MA sigs))
                       '(LSU rbuf))
                (wbuf0-stg-p (INST-stg (step-INST j MT MA sigs)))
                (wbuf1-stg-p (INST-stg j))
                (INST-in i MT) (INST-p i)
                (INST-in j MT) (INST-p j)
                (not (b1p (rbuf-wbuf0? (LSU-rbuf (step-LSU MA sigs))))))
           (INST-in-order-p i j MT))
  :hints (("Goal" :use ((:instance stages-reachable-to-LSU-rbuf))
                  :restrict ((LSU-RBUF-VALID-IF-INST-IN ((i i)))
                             (INST-IN-ORDER-P-LSU-ISSUED-RS ((i j) (j i)))
                             (INST-IN-ORDER-P-RBUF-WBUF0 ((i i) (j j)))
                             (wbuf1-not-advance-if-rbuf-inst-in ((i i))))
                  :in-theory (e/d (INST-IN-ORDER-P-LSU-ISSUED-RS
                                   INST-IN-ORDER-P-RBUF-WBUF0)
                                  ()))))

(defthm in-order-rbuf-wbuf0-step-inst
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (equal (INST-stg (step-INST i MT MA sigs))
                       '(LSU rbuf))
                (wbuf0-stg-p (INST-stg (step-INST j MT MA sigs)))
                (INST-in i MT) (INST-p i)
                (INST-in j MT) (INST-p j)
                (not (b1p (rbuf-wbuf0? (LSU-rbuf (step-LSU MA sigs))))))
           (INST-in-order-p i j MT))
  :hints (("Goal" :cases ((wbuf0-stg-p (INST-stg j))
                          (wbuf1-stg-p (INST-stg j)))
                  :in-theory (enable wbuf-stg-p*)))
  :rule-classes
  ((:rewrite :corollary
     (implies (and (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (equal (INST-stg (step-INST i MT MA sigs))
                          '(LSU rbuf))
                   (INST-in i MT) (INST-p i)
                   (INST-in j MT) (INST-p j)
                   (INST-in-order-p j i MT)
                   (not (b1p (rbuf-wbuf0? (LSU-rbuf (step-LSU MA sigs))))))
              (not (wbuf0-stg-p (INST-stg (step-INST j MT MA sigs))))))))

(encapsulate nil
(local
(defthm in-order-rbuf-wbuf0-step-trace-help
    (implies (and (inv MT MA)
```

```
                  (wbuf0-stg-p (INST-stg (step-INST i MT MA sigs)))
                  (INST-in i MT) (INST-p i)
                  (subtrace-p trace MT) (INST-listp trace)
                  (subtrace-after-p i trace MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (rbuf-valid? (LSU-rbuf (step-LSU MA sigs))))
                  (not (b1p (rbuf-wbuf0? (LSU-rbuf (step-LSU MA sigs)))))
                  (b1p (wbuf-valid? (LSU-wbuf0 (step-LSU MA sigs)))))
     (no-inst-at-stg-in-trace '(LSU rbuf)
                              (step-trace trace MT MA sigs ISA spc smc)))))

(defthm in-order-rbuf-wbuf0-step-trace
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (rbuf-valid? (LSU-rbuf (step-LSU MA sigs))))
                  (not (b1p (rbuf-wbuf0? (LSU-rbuf (step-LSU MA sigs)))))
                  (b1p (wbuf-valid? (LSU-wbuf0 (step-LSU MA sigs)))))
     (in-order-rbuf-wbuf0-p (step-trace trace MT MA sigs ISA spc smc))))
)

(defthm LSU-rbuf-wbuf1-step-LSU
    (implies (and (b1p (rbuf-valid? (LSU-rbuf (MA-LSU MA))))
                  (not (b1p (release-rbuf? (MA-LSU MA) MA sigs))))
             (equal (rbuf-wbuf1? (LSU-rbuf (step-LSU MA sigs)))
                    (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA)))))
  :hints (("Goal" :in-theory (enable step-LSU step-rbuf lift-b-ops))))

(defthm INST-in-order-wbuf1-rbuf-step-inst-help1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (not (wbuf1-stg-p (INST-stg i)))
                  (equal (INST-stg (step-inst j MT MA sigs))
                         '(LSU rbuf))
                  (wbuf1-stg-p (INST-stg (step-INST i MT MA sigs)))
                  (b1p (rbuf-wbuf1? (LSU-rbuf (step-LSU MA sigs)))))
             (INST-in-order-p i j MT))
  :hints (("Goal" :use ((:instance stages-reachable-to-LSU-rbuf
                                   (i j))
                        (:instance not-wbuf1-stg-p-step-inst-if-not-LSU-RS
                                   (i i)))
                  :in-theory (e/d (INST-IN-ORDER-P-LSU-ISSUED-RS
                                   lift-b-ops ISSUE-LSU-RS0?
                                   ISSUE-LSU-RS1?
                                   LSU-RS0-ISSUE-READY?
                                   LSU-RS1-ISSUE-READY?
                                   RELEASE-WBUF0?
                                   inst-stg-step-inst)
                                  (not-wbuf0-stg-p-step-inst-if-not-LSU-RS)))))

(defthm INST-in-order-wbuf1-rbuf-step-inst-help2
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (MAETT-p MT) (MA-state-p MA)
                  (wbuf1-stg-p (INST-stg i))
                  (equal (INST-stg (step-inst j MT MA sigs))
                         '(LSU rbuf))
                  (wbuf1-stg-p (INST-stg (step-INST i MT MA sigs)))
                  (b1p (rbuf-wbuf1? (LSU-rbuf (step-LSU MA sigs)))))
```

```
            (INST-in-order-p i j MT))
  :hints (("Goal" :use ((:instance stages-reachable-to-LSU-rbuf
                                   (i j)))
                  :in-theory (enable inst-stg-step-inst WBUF1-STG-P
                                     INST-IN-ORDER-P-WBUF1-RBUF
                                     INST-IN-ORDER-P-LSU-ISSUED-RS))))

(defthm INST-in-order-wbuf1-rbuf-step-inst
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (equal (INST-stg (step-inst j MT MA sigs))
                         '(LSU rbuf))
                  (wbuf1-stg-p (INST-stg (step-INST i MT MA sigs)))
                  (b1p (rbuf-wbuf1? (LSU-rbuf (step-LSU MA sigs)))))
             (INST-in-order-p i j MT))
  :hints (("Goal" :cases ((wbuf1-stg-p (INST-stg i)))
                  :in-theory (enable wbuf-stg-p*)))
  :rule-classes
  ((:rewrite :corollary
     (implies (and (inv MT MA)
                   (equal (INST-stg (step-inst j MT MA sigs))
                          '(LSU rbuf))
                   (INST-in i MT) (INST-p i)
                   (INST-in j MT) (INST-p j)
                   (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                   (INST-in-order-p j i MT)
                   (b1p (rbuf-wbuf1? (LSU-rbuf (step-LSU MA sigs)))))
              (not (wbuf1-stg-p (INST-stg (step-INST i MT MA sigs))))))))

(encapsulate nil
(local
(defthm in-order-wbuf1-rbuf-step-trace-help
    (implies (and (inv MT MA)
                  (equal (INST-stg (step-inst i MT MA sigs))
                         '(LSU rbuf))
                  (INST-in i MT) (INST-p i)
                  (subtrace-p trace MT) (INST-listp trace)
                  (subtrace-after-p i trace MT)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (b1p (rbuf-valid? (LSU-rbuf (step-LSU MA sigs))))
                  (b1p (rbuf-wbuf1? (LSU-rbuf (step-LSU MA sigs))))
                  (b1p (wbuf-valid? (LSU-wbuf1 (step-LSU MA sigs)))))
             (no-inst-at-wbuf1-p (step-trace trace MT MA sigs ISA spc smc)))))

(defthm in-order-wbuf1-rbuf-step-trace
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (b1p (rbuf-valid? (LSU-rbuf (step-LSU MA sigs))))
                  (b1p (rbuf-wbuf1? (LSU-rbuf (step-LSU MA sigs))))
                  (b1p (wbuf-valid? (LSU-wbuf1 (step-LSU MA sigs)))))
     (in-order-wbuf1-rbuf-p (step-trace trace MT MA sigs ISA spc smc))))
)

(defthm INST-in-order-rbuf-wbuf1-step-INST-help1
    (implies (and (inv MT MA)
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU rbuf))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
```

```
                (MAETT-p MT) (MA-state-p MA)
                (not (wbuf1-stg-p (INST-stg j)))
                (wbuf1-stg-p (INST-stg (step-INST j MT MA sigs)))
                (not (b1p (rbuf-wbuf1? (LSU-rbuf (step-LSU MA sigs))))))
           (INST-in-order-p i j MT))
  :hints (("Goal"  :use ((:instance stages-reachable-to-LSU-rbuf
                                    (i i))
                         (:instance not-wbuf1-stg-p-step-inst-if-not-LSU-RS
                                    (i j)))
                  :in-theory (e/d
                              (INST-IN-ORDER-P-LSU-ISSUED-RS)
                              (not-wbuf1-stg-p-step-inst-if-not-LSU-RS)))))

(defthm LSU-rbuf-wbuf1-step-LSU-if-wbuf1-not-advance
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU rbuf))
                  (wbuf1-stg-p (INST-stg (step-INST j MT MA sigs)))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (wbuf1-stg-p (INST-stg J))
                  (or (equal (INST-stg i) '(LSU RS0))
                      (equal (INST-stg i) '(LSU RS1))))
             (equal (rbuf-wbuf1? (LSU-rbuf (step-LSU MA sigs))) 1))
  :hints (("Goal" :in-theory (enable step-LSU step-rbuf lift-b-ops
                                     ISSUE-LSU-RS0? ISSUE-LSU-RS1?
                                     RELEASE-WBUF0?
                                     wbuf1-stg-p inst-stg-step-inst))))

(defthm LSU-rbuf-wbuf1-step-LSU-if-rbuf-not-released
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU rbuf))
                  (wbuf1-stg-p (INST-stg (step-INST j MT MA sigs)))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (wbuf1-stg-p (INST-stg J))
                  (equal (INST-stg i) '(LSU rbuf)))
             (equal (rbuf-wbuf1? (LSU-rbuf (step-LSU MA sigs)))
                    (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA)))))
  :hints (("Goal" :in-theory (enable step-LSU step-rbuf lift-b-ops
                                     LSU-RBUF-VALID-IF-INST-IN
                                     inst-stg-step-inst wbuf1-stg-p)
                  :restrict ((LSU-RBUF-VALID-IF-INST-IN ((i i)))))))

(defthm INST-in-order-rbuf-wbuf1-step-INST-help2
    (implies (and (inv MT MA)
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU rbuf))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (MAETT-p MT) (MA-state-p MA)
                  (wbuf1-stg-p (INST-stg j))
                  (wbuf1-stg-p (INST-stg (step-INST j MT MA sigs)))
                  (not (b1p (rbuf-wbuf1? (LSU-rbuf (step-LSU MA sigs))))))
             (INST-in-order-p i j MT))
  :hints (("Goal" :use ((:instance stages-reachable-to-LSU-rbuf
                                   (i i)))
                  :in-theory (enable
                              INST-IN-ORDER-P-RBUF-WBUF1
```

```
                              INST-IN-ORDER-P-LSU-ISSUED-RS))))

(defthm INST-in-order-rbuf-wbuf1-step-INST
    (implies (and (inv MT MA)
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU rbuf))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (MAETT-p MT) (MA-state-p MA)
                  (wbuf1-stg-p (INST-stg (step-INST j MT MA sigs)))
                  (not (b1p (rbuf-wbuf1? (LSU-rbuf (step-LSU MA sigs))))))
             (INST-in-order-p i j MT))
  :hints (("Goal" :cases ((wbuf1-stg-p (INST-stg j)))))
  :rule-classes
  ((:rewrite :corollary
     (implies (and (inv MT MA)
                   (equal (INST-stg (step-INST i MT MA sigs))
                          '(LSU rbuf))
                   (INST-in i MT) (INST-p i)
                   (INST-in j MT) (INST-p j)
                   (INST-in-order-p j i MT)
                   (MAETT-p MT) (MA-state-p MA)
                   (not (b1p (rbuf-wbuf1? (LSU-rbuf (step-LSU MA sigs))))))
              (not (wbuf1-stg-p (INST-stg (step-INST j MT MA sigs))))))))

(encapsulate nil
(local
(defthm in-order-rbuf-wbuf1-step-trace-help
    (implies (and (inv MT MA)
                  (wbuf1-stg-p (INST-stg (step-INST i MT MA sigs)))
                  (INST-in i MT) (INST-p i)
                  (subtrace-p trace MT) (INST-listp trace)
                  (subtrace-after-p i trace MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (rbuf-valid? (LSU-rbuf (step-LSU MA sigs))))
                  (not (b1p (rbuf-wbuf1? (LSU-rbuf (step-LSU MA sigs)))))
                  (b1p (wbuf-valid? (LSU-wbuf1 (step-LSU MA sigs)))))
     (no-inst-at-stg-in-trace '(LSU rbuf)
                              (step-trace trace MT MA sigs ISA spc smc)))))

(defthm in-order-rbuf-wbuf1-step-trace
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (rbuf-valid? (LSU-rbuf (step-LSU MA sigs))))
                  (not (b1p (rbuf-wbuf1? (LSU-rbuf (step-LSU MA sigs)))))
                  (b1p (wbuf-valid? (LSU-wbuf1 (step-LSU MA sigs)))))
     (in-order-rbuf-wbuf1-p (step-trace trace MT MA sigs ISA spc smc))))
)

(defthm in-order-load-store-p-mt-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (in-order-load-store-p (MT-step MT MA sigs) (MA-step MA sigs)))
  :hints (("Goal" :in-theory (enable in-order-load-store-p))))

(in-theory (disable INST-IN-ORDER-RBUF-WBUF1-STEP-INST-HELP2
                    INST-IN-ORDER-RBUF-WBUF1-STEP-INST-HELP1
                    INST-IN-ORDER-WBUF1-RBUF-STEP-INST-HELP2
                    INST-IN-ORDER-WBUF1-RBUF-STEP-INST-HELP1
                    IN-ORDER-RBUF-WBUF0-STEP-INST-HELP3
                    IN-ORDER-RBUF-WBUF0-STEP-INST-HELP2
```

```
                    IN-ORDER-RBUF-WBUF0-STEP-INST-HELP1
                    INST-IN-ORDER-WBUF0-RBUF-STEP-INST-HELP3
                    INST-IN-ORDER-WBUF0-RBUF-STEP-INST-HELP2
                    INST-IN-ORDER-WBUF0-RBUF-STEP-INST-HELP1
                    INST-IN-ORDER-WBUF0-WBUF1-STEP-INST-HELP2))

(in-theory (disable INST-IN-ORDER-INST-OF-TAG-IF-ROB-FLG
                    INST-IN-ORDER-INST-OF-TAG-IF-NOT-ROB-FLG
                    INST-IN-ORDER-INST-OF-TAG-IF-TAG-IN-ORDER
                    INST-IN-ORDER-INST-OF-TAG-IF-GT-ROB-HEAD
                    INST-IN-ORDER-INST-OF-TAG-IF-LE-ROB-TAIL
                    INST-IN-ORDER-P-TOTAL
                    DE-VALID-CONSISTENT))

(defthm INST-in-order-p-step-INST-LSU-issued-RS0-help2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg j) '(LSU RS0))
                  (LSU-stg-p (INST-stg i))
                  (equal (INST-stg (step-INST j MT MA sigs)) '(LSU RS0))
                  (LSU-issued-stg-p (INST-stg (step-INST i MT MA sigs)))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j))
             (INST-in-order-p i j MT))
  :hints (("Goal" :in-theory (enable LSU-stg-p LSU-issued-stg-p
                                     INST-IN-ORDER-P-LSU-ISSUED-RS
                                     INST-in-order-p-LSU-RS1-RS0
                                     ISSUE-LSU-RS0? lift-b-ops
                                     LSU-RS0-ISSUE-READY?
                                     ISSUE-LSU-RS1?
                                     LSU-RS1-ISSUE-READY?
                                     inst-stg-step-inst))))

(defthm INST-in-order-p-step-INST-LSU-issued-RS0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg (step-INST j MT MA sigs)) '(LSU RS0))
                  (LSU-issued-stg-p (INST-stg (step-INST i MT MA sigs)))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j))
             (INST-in-order-p i j MT))
  :hints (("Goal" :use ((:instance reachable-stages-to-LSU-issued-stg-p)
                        (:instance stages-reachable-to-LSU-RS0 (i j)))
                  :in-theory (e/d (LSU-stg-p INST-IN-ORDER-P-LSU-ISSUED-RS
                                             LSU-issued-stg-p)
                                  (INST-STG-STEP-IFU-INST-IF-DQ-FULL))))
  :rule-classes
  ((:rewrite :corollary
     (implies (and (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (equal (INST-stg (step-INST j MT MA sigs)) '(LSU RS0))
                   (INST-in i MT) (INST-p i)
                   (INST-in j MT) (INST-p j)
                   (INST-in-order-p j i MT))
              (not (LSU-issued-stg-p (INST-stg (step-INST i MT MA sigs))))))))

(defthm INST-in-order-p-step-INST-LSU-issued-RS1-help2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg j) '(LSU RS1))
                  (LSU-stg-p (INST-stg i))
                  (equal (INST-stg (step-INST j MT MA sigs)) '(LSU RS1))
```

```
                (LSU-issued-stg-p (INST-stg (step-INST i MT MA sigs)))
                (INST-in i MT) (INST-p i)
                (INST-in j MT) (INST-p j))
           (INST-in-order-p i j MT))
  :hints (("Goal" :in-theory (enable LSU-stg-p LSU-issued-stg-p
                                     INST-IN-ORDER-P-LSU-ISSUED-RS
                                     INST-in-order-p-LSU-RS0-RS1
                                     ISSUE-LSU-RS0? lift-b-ops
                                     LSU-RS0-ISSUE-READY?
                                     ISSUE-LSU-RS1?
                                     LSU-RS1-ISSUE-READY?
                                     inst-stg-step-inst))))

(defthm INST-in-order-p-step-INST-LSU-issued-RS1
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (equal (INST-stg (step-INST j MT MA sigs)) '(LSU RS1))
                (LSU-issued-stg-p (INST-stg (step-INST i MT MA sigs)))
                (INST-in i MT) (INST-p i)
                (INST-in j MT) (INST-p j))
           (INST-in-order-p i j MT))
  :hints (("Goal" :use ((:instance reachable-stages-to-LSU-issued-stg-p)
                        (:instance stages-reachable-to-LSU-RS1 (i j)))
                :in-theory (e/d (LSU-stg-p INST-IN-ORDER-P-LSU-ISSUED-RS
                                           LSU-issued-stg-p)
                                (INST-STG-STEP-IFU-INST-IF-DQ-FULL))))

  :rule-classes
  ((:rewrite :corollary
     (implies (and (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (equal (INST-stg (step-INST j MT MA sigs)) '(LSU RS1))
                   (INST-in i MT) (INST-p i)
                   (INST-in j MT) (INST-p j)
                   (INST-in-order-p j i MT))
              (not (LSU-issued-stg-p (INST-stg (step-INST i MT MA sigs)))))))))

(encapsulate nil
(local
(defthm in-order-LSU-issue-p-step-trace-help1
    (implies (and (inv MT MA)
                (equal (INST-stg (step-INST i MT MA sigs))
                       '(LSU RS0))
                (INST-in i MT) (INST-p i)
                (subtrace-p trace MT) (INST-listp trace)
                (subtrace-after-p i trace MT)
                (MAETT-p MT) (MA-state-p MA))
     (no-issued-LSU-inst-p (step-trace trace MT MA sigs ISA spc smc)))))

(local
(defthm in-order-LSU-issue-p-step-trace-help2
    (implies (and (inv MT MA)
                (equal (INST-stg (step-INST i MT MA sigs))
                       '(LSU RS1))
                (INST-in i MT) (INST-p i)
                (subtrace-p trace MT) (INST-listp trace)
                (subtrace-after-p i trace MT)
                (MAETT-p MT) (MA-state-p MA))
     (no-issued-LSU-inst-p (step-trace trace MT MA sigs ISA spc smc)))))

(defthm in-order-LSU-issue-p-step-trace
    (implies (and (inv MT MA)
```

```
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA))
           (in-order-LSU-issue-p (step-trace trace MT MA sigs ISA spc smc))))
)

(defthm INST-in-order-RS1-RS0-if-RS1-head
    (implies (and (inv MT MA)
                (equal (INST-stg (step-INST i MT MA sigs))
                       '(LSU RS1))
                (equal (INST-stg (step-INST j MT MA sigs))
                       '(LSU RS0))
                (INST-in i MT) (INST-p i)
                (INST-in j MT) (INST-p j)
                (b1p (LSU-RS1-head? (step-LSU MA sigs)))
                (MAETT-p MT) (MA-state-p MA))
           (INST-in-order-p i j MT))
  :hints (("goal" :use ((:instance stages-reachable-to-LSU-RS1)
                        (:instance stages-reachable-to-LSU-RS0
                                   (i j)))
                :in-theory (enable step-inst-dq-inst step-inst-execute-inst
                                   step-inst-low-level-functions
                                   INST-in-order-p-LSU-RS1-RS0
                                   lift-b-ops LSU-READY?
                                   DISPATCH-TO-LSU? STEP-LSU
                                   step-RS1-head?
                                   DISPATCH-INST?)))
  :rule-classes
  ((:rewrite :corollary
           (implies (and (inv MT MA)
                         (equal (INST-stg (step-INST j MT MA sigs))
                                '(LSU RS0))
                         (INST-in i MT) (INST-p i)
                         (INST-in j MT) (INST-p j)
                         (INST-in-order-p j i MT)
                         (b1p (LSU-RS1-head? (step-LSU MA sigs)))
                         (MAETT-p MT) (MA-state-p MA))
                    (not (equal (INST-stg (step-INST i MT MA sigs))
                                '(LSU RS1)))))))

(defthm INST-in-order-RS0-RS1-if-not-RS1-head
    (implies (and (inv MT MA)
                (equal (INST-stg (step-INST j MT MA sigs))
                       '(LSU RS1))
                (equal (INST-stg (step-INST i MT MA sigs))
                       '(LSU RS0))
                (INST-in i MT) (INST-p i)
                (INST-in j MT) (INST-p j)
                (not (b1p (LSU-RS1-head? (step-LSU MA sigs))))
                (MAETT-p MT) (MA-state-p MA))
           (INST-in-order-p i j MT))
  :hints (("goal" :use ((:instance stages-reachable-to-LSU-RS1 (i j))
                        (:instance stages-reachable-to-LSU-RS0))
                :in-theory (enable step-inst-dq-inst step-inst-execute-inst
                                   step-inst-low-level-functions
                                   INST-in-order-p-LSU-RS0-RS1
                                   lift-b-ops LSU-READY?
                                   DISPATCH-TO-LSU? STEP-LSU
                                   step-RS1-head?
                                   DISPATCH-INST?)))
  :rule-classes
  ((:rewrite :corollary
           (implies (and (inv MT MA)
```

```
                        (equal (INST-stg (step-INST j MT MA sigs))
                               '(LSU RS1))
                        (INST-in i MT) (INST-p i)
                        (INST-in j MT) (INST-p j)
                        (INST-in-order-p j i MT)
                        (not (b1p (LSU-RS1-head? (step-LSU MA sigs))))
                        (MAETT-p MT) (MA-state-p MA))
                   (not (equal (INST-stg (step-INST i MT MA sigs))
                               '(LSU RS0)))))))

(encapsulate nil
(local
(defthm in-order-LSU-RS-p-MT-trace-MT-step-help1
    (implies (and (inv MT MA)
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU RS0))
                  (INST-in i MT) (INST-p i)
                  (subtrace-p trace MT) (INST-listp trace)
                  (subtrace-after-p i trace MT)
                  (b1p (LSU-RS1-head? (step-LSU MA sigs)))
                  (MAETT-p MT) (MA-state-p MA))
             (no-inst-at-stg-in-trace '(LSU RS1)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
(defthm in-order-LSU-RS-p-MT-trace-MT-step-help2
    (implies (and (inv MT MA)
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU RS1))
                  (INST-in i MT) (INST-p i)
                  (subtrace-p trace MT) (INST-listp trace)
                  (subtrace-after-p i trace MT)
                  (not (b1p (LSU-RS1-head? (step-LSU MA sigs))))
                  (MAETT-p MT) (MA-state-p MA))
             (no-inst-at-stg-in-trace '(LSU RS0)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(defthm in-order-LSU-RS-p-MT-trace-MT-step
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA))
             (in-order-LSU-RS-p (step-trace trace MT MA sigs ISA spc smc)
                                (MA-step MA sigs))))
)

(defthm not-wbuf0-step-inst-if-rob-wbuf-empty
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (b1p (ROB-empty? (MA-rob MA)))
                  (not (b1p (LSU-pending-writes? (MA-LSU MA))))
                  (MAETT-p MT) (MA-state-p MA))
             (not (wbuf0-stg-p (INST-stg (step-inst i MT MA sigs)))))
  :hints (("Goal" :use ((:instance stages-reachable-to-LSU-wbuf0)
                        (:instance stages-reachable-to-LSU-wbuf0-lch)
                        (:instance stages-reachable-to-complete-wbuf0)
                        (:instance stages-reachable-to-commit-wbuf0))
                  :in-theory (enable wbuf0-stg-p)
                  :restrict (((:REWRITE NOT-ROB-EMPTY-IF-INST-IS-EXECUTED . 1)
                              ((i i)))))))
```

```
(defthm not-INST-commit-if-LSU-wbuf0
    (implies (and (inv MT MA)
                  (equal (INST-stg j) '(LSU wbuf0))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (equal (INST-stg i) '(complete wbuf1))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-commit? i MA) 0))
  :hints (("goal" :in-theory (e/d (INST-commit? lift-b-ops
                                                equal-b1p-converter
                                                INST-IN-ORDER-P-WBUF0-WBUF1)
                                  (UNCOMMITTED-INST-P-IS-AFTER-MT-ROB-HEAD))
                  :use (:instance UNCOMMITTED-INST-P-IS-AFTER-MT-ROB-HEAD
                                  (i j)))))

(defthm INST-in-order-if-INST-commit
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (execute-stg-p (INST-stg j))
                  (b1p (INST-commit? i MA)))
             (INST-in-order-p i j MT))
  :hints (("Goal" :in-theory (e/d (INST-commit? lift-b-ops)
                                  (UNCOMMITTED-INST-P-IS-AFTER-MT-ROB-HEAD))
                  :use (:instance UNCOMMITTED-INST-P-IS-AFTER-MT-ROB-HEAD
                                  (i j)))))

(defthm INST-in-order-if-INST-commit-before-complete
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (not (equal i j))
                  (complete-stg-p (INST-stg j))
                  (b1p (INST-commit? i MA)))
             (INST-in-order-p i j MT))
  :hints (("Goal" :in-theory (e/d (INST-commit? lift-b-ops)
                                  (UNCOMMITTED-INST-P-IS-AFTER-MT-ROB-HEAD))
                  :use (:instance UNCOMMITTED-INST-P-IS-AFTER-MT-ROB-HEAD
                                  (i j)))))

(defthm INST-in-order-retire-LSU-wbuf0-step-inst
    (implies (and (inv MT MA)
                  (retire-stg-p (INST-stg (step-INST i MT MA sigs)))
                  (equal (INST-stg (step-inst j MT MA sigs))
                         '(LSU wbuf0))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (b1p (INST-store? i))
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (MAETT-p MT) (MA-state-p MA))
             (INST-in-order-p i j MT))
  :hints (("Goal" :use ((:instance stages-reachable-to-retire-stg)
                        (:instance stages-reachable-to-LSU-wbuf0
                                   (i j))
                        (:instance uniq-wbuf0-inst)
                        (:instance uniq-wbuf1-inst))
                  :in-theory (enable wbuf0-stg-p
                                     INST-stg-step-inst lift-b-ops
                                     NOT-INST-STORE-IF-COMPLETE
                                     INST-IN-ORDER-P-RETIRE-WBUF1
```

```
                                    INST-IN-ORDER-P-RETIRE-WBUF0
                                    INST-IN-ORDER-P-LSU-ISSUED-RS
                                    INST-IN-ORDER-P-WBUF0-WBUF1))))

(defthm INST-in-order-retire-LSU-wbuf0-lch-step-inst
    (implies (and (inv MT MA)
                  (retire-stg-p (INST-stg (step-INST i MT MA sigs)))
                  (equal (INST-stg (step-inst j MT MA sigs))
                         '(LSU wbuf0 lch))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (b1p (INST-store? i))
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (INST-in-order-p i j MT))
  :hints (("Goal" :use ((:instance stages-reachable-to-retire-stg)
                        (:instance stages-reachable-to-LSU-wbuf0-lch
                                   (i j))
                        (:instance uniq-wbuf0-inst)
                        (:instance uniq-wbuf1-inst))
                  :in-theory (enable wbuf0-stg-p
                                     INST-stg-step-inst lift-b-ops
                                     NOT-INST-STORE-IF-COMPLETE
                                     INST-IN-ORDER-P-RETIRE-WBUF1
                                     INST-IN-ORDER-P-RETIRE-WBUF0
                                     INST-IN-ORDER-P-LSU-ISSUED-RS
                                     INST-IN-ORDER-P-WBUF0-WBUF1))))

(defthm INST-in-order-retire-complete-wbuf0-lch-step-inst
    (implies (and (inv MT MA)
                  (retire-stg-p (INST-stg (step-INST i MT MA sigs)))
                  (equal (INST-stg (step-inst j MT MA sigs))
                         '(complete wbuf0))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (b1p (INST-store? i))
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (INST-in-order-p i j MT))
  :hints (("Goal" :use
                  ((:instance stages-reachable-to-retire-stg)
                   (:instance stages-reachable-to-complete-wbuf0
                              (i j))
                   (:instance INST-in-order-if-INST-commit-before-complete)
                   (:instance uniq-wbuf0-inst)
                   (:instance uniq-wbuf1-inst))
                  :in-theory (enable wbuf0-stg-p
                                     INST-stg-step-inst lift-b-ops
                                     NOT-INST-STORE-IF-COMPLETE
                                     INST-IN-ORDER-P-RETIRE-WBUF1
                                     INST-IN-ORDER-P-RETIRE-WBUF0
                                     INST-IN-ORDER-P-LSU-ISSUED-RS
                                     INST-IN-ORDER-P-WBUF0-WBUF1))))

(encapsulate nil
(defthm not-INST-commit-if-store-inst-is-at-complete-help
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(complete))
                  (b1p (INST-store? i))
                  (b1p (LSU-pending-writes? (MA-LSU MA)))
                  (not (b1p (inst-specultv? i)))
```

```
                (not (b1p (INST-modified? i)))
                (MAETT-p MT) (MA-state-p MA))
           (equal (INST-commit? i MA) 0))
  :hints (("Goal" :cases ((INST-fetch-error-detected-p i)
                          (INST-decode-error-detected-p i))
                  :in-theory (enable NOT-INST-STORE-IF-COMPLETE
                                     INST-COMMIT? lift-b-ops
                                     equal-b1p-converter
                                     INST-EXCPT-DETECTED-P
                                     commit-inst?))))

(defthm not-INST-commit-if-store-inst-is-at-complete
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(complete))
                  (b1p (INST-store? i))
                  (b1p (LSU-pending-writes? (MA-LSU MA)))
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (equal (INST-commit? i MA) 0))
  :hints (("Goal" :cases ((b1p (inst-specultv? i))
                          (b1p (INST-modified? i)))
                  :in-theory (enable equal-b1p-converter))))
)

(defthm not-enter-excpt-if-LSU-pending-writes
    (implies (b1p (LSU-pending-writes? (MA-LSU MA)))
             (equal (enter-excpt? MA) 0))
  :hints (("Goal" :in-theory (enable enter-excpt? lift-b-ops
                                     equal-b1p-converter))))

(defthm INST-in-order-retire-commit-wbuf0-1ch-step-inst
    (implies (and (inv MT MA)
                  (retire-stg-p (INST-stg (step-INST i MT MA sigs)))
                  (equal (INST-stg (step-inst j MT MA sigs))
                         '(commit wbuf0))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (b1p (INST-store? i))
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (INST-in-order-p i j MT))
  :hints (("Goal" :use
                  ((:instance stages-reachable-to-retire-stg)
                   (:instance stages-reachable-to-commit-wbuf0
                              (i j))
                   (:instance INST-in-order-if-INST-commit-before-complete)
                   (:instance uniq-wbuf0-inst)
                   (:instance uniq-wbuf1-inst))
                  :restrict ((LSU-pending-writes-if-wbuf-inst-in
                              ((i j))))
                  :in-theory (enable wbuf0-stg-p
                                     INST-stg-step-inst lift-b-ops
                                     NOT-INST-STORE-IF-COMPLETE
                                     INST-IN-ORDER-P-RETIRE-WBUF1
                                     INST-IN-ORDER-P-RETIRE-WBUF0
                                     INST-IN-ORDER-P-LSU-ISSUED-RS
                                     INST-IN-ORDER-P-WBUF0-WBUF1))))

(defthm INST-in-order-retire-wbuf0-step-inst
    (implies (and (inv MT MA)
                  (retire-stg-p (INST-stg (step-INST i MT MA sigs)))
```

```
                (wbuf0-stg-p (INST-stg (step-inst j MT MA sigs)))
                (INST-in i MT) (INST-p i)
                (INST-in j MT) (INST-p j)
                (b1p (INST-store? i))
                (not (MT-CMI-p (MT-step MT MA sigs)))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (INST-in-order-p i j MT))
  :hints (("Goal" :in-theory (enable wbuf0-stg-p)))
  :rule-classes
  ((:rewrite :corollary
     (implies (and (inv MT MA)
                   (wbuf0-stg-p (INST-stg (step-inst j MT MA sigs)))
                   (INST-in i MT) (INST-p i)
                   (INST-in j MT) (INST-p j)
                   (INST-in-order-p j i MT)
                   (not (MT-CMI-p (MT-step MT MA sigs)))
                   (b1p (INST-store? i))
                   (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
              (not (retire-stg-p (INST-stg (step-INST i MT MA sigs))))))))

(encapsulate nil
(local
(defthm in-order-wb-retire-p-MT-trace-MT-step-help
    (implies (and (inv MT MA)
                  (wbuf0-stg-p (INST-stg (step-inst i MT MA sigs)))
                  (INST-in i MT) (INST-p i)
                  (subtrace-p trace MT) (INST-listp trace)
                  (subtrace-after-p i trace MT)
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (no-retired-store-p (step-trace trace MT MA sigs ISA spc smc)))
  :hints (("Goal" :in-theory (enable INST-exintr-now? lift-b-ops
                                     ex-intr?)))))

(defthm in-order-wb-retire-p-MT-trace-MT-step
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (in-order-wb-retire-p (step-trace trace MT MA sigs ISA spc smc))))
)

(defthm in-order-LSU-inst-p-preserved
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (not (MT-CMI-p (MT-step MT MA sigs))))
             (in-order-LSU-inst-p (MT-step MT MA sigs) (MA-step MA sigs)))
  :hints (("Goal" :in-theory (e/d (in-order-LSU-inst-p) ()))))
```

D.6.5 MI-inv.lisp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MI-inv.lisp
; Author  Jun Sawada, University of Texas at Austin
;
;  This book contains the proof of the invariant property MT-inst-inv.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(in-package "ACL2")

(include-book "MA2-lemmas")
```

```
(include-book "MAETT-lemmas")
(include-book "modifier")
(include-book "memory-inv")
(deflabel begin-MI-inv)
; This file proves instruction invariants.
; Index
;  Misc Lemmas
;  Proof of MI-INST-inv for initial states
;  Proof of MI-INST-inv for induction step
;     Proof of INST-inv-fetched-inst
;     Proof of INST-inv-step-INST
;        Proof of IFU-inst-inv
;        Proof of DQ-inst-inv
;        Proof of execute-inst-inv-step-INST
;          Lemmas about the stage of dispatched instructions.
;          Lemmas about dispatch logic
;          Lemmas about register modifiers
;          Lemmas about CDB output
;          Proof of IU-inst-inv-step-INST
;          Proof of MU-inst-inv-step-INST
;          Proof of BU-inst-inv-step-INST
;          Proof of LSU-inst-inv-step-INST
;             LSU-RS0-inst-inv-step-INST
;             LSU-RS1-inst-inv-step-INST
;             LSU-RS1-inst-inv-step-INST
;             LSU-wbuf0-inst-inv-step-INST
;             LSU-rbuf-inst-inv-step-INST
;             LSU-wbuf0-lch-inst-inv-step-INST
;             LSU-wbuf1-lch-inst-inv-step-INST
;               LSU-forward-wbuf-INST-dest-val
;               read-mem-INST-load-addr-INST-dest-val
;             LSU-lch-inst-inv-step-INST
;          Proof of execute-inst-robe-inv-step-INST
;        Proof of complete-inst-inv
;        Proof of commit-inst-inv
;     Proof of INST-inv-exintr-INST
;   MT-INST-inv-lemma
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Misc Lemmas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Several lemmas given below are for the proof of lemmas in this book.
;; They are not intended to be universal lemmas, because the left-hand
;; side terms are not definitely more complex than right-hand side.
;; However, these rewriting rules are useful in the proof of lemmas in
;; this book.

(in-theory (enable MT-specultv-at-dispatch-off-if-non-specultv-inst-in
                   INST-modified-at-dispatch-off-if-undispatched-inst-in))

(local
(defthm MT-mem-=-MA-mem
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA))
             (equal (MT-mem MT) (MA-mem MA)))
  :hints (("goal" :in-theory (enable weak-inv inv
                                     mem-match-p)))))

(local
(defthm MT-RF-=-MA-RF
    (implies (and (inv MT MA)
                  (MAETT-p MT)
```

```
                (MA-state-p MA))
           (equal (MT-RF MT) (MA-RF MA)))
  :hints (("goal" :in-theory (enable weak-inv inv
                                     RF-match-p)))))

(local
(defthm MT-SRF-=-MA-SRF
    (implies (and (inv MT MA)
                (MAETT-p MT)
                (MA-state-p MA))
           (equal (MT-SRF MT) (MA-SRF MA)))
  :hints (("goal" :in-theory (enable weak-inv inv
                                     SRF-match-p)))))

(in-theory (enable ISA-before-MT-non-nil-trace))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Proof of MI-INST-inv for initial states
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defthm Mt-INST-inv-init-MT
    (MT-INST-inv (init-MT MA) MA)
  :hints (("goal" :in-theory (enable MT-INST-inv init-MT))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Proof of MI-INST-inv for induction step
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Proof of Invariants on each instruction
;   Most of the effort in this book is directed for the proof of
;   following lemmas:
;     INST-inv-fetched-inst
;     INST-inv-step-INST
;     INST-inv-exintr-INST
;   And the proof of INST-inv-step-INST takes by far the
;   largest number of lemmas..
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Begin of INST-inv-fetched-inst
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(encapsulate nil
(local
(defthm ISA-pc-MT-final-ISA-help-help
    (implies (and (equal (trace-pc trace (ISA-pc pre)) (MA-pc MA))
                (MAETT-p MT)
                (MA-state-p MA))
           (equal (ISA-pc (trace-final-ISA trace pre)) (MA-PC MA)))))

(local
(defthm ISA-pc-MT-final-ISA-help
    (implies (and (equal (MT-pc MT) (MA-pc MA))
                (MAETT-p MT)
                (MA-state-p MA))
           (equal (ISA-pc (MT-final-ISA MT)) (MA-PC MA)))
  :hints (("goal" :in-theory (enable MT-final-ISA MT-pc)))))

;; The program counter in a current MA is the same as the final ISA state.
(defthm ISA-pc-MT-final-ISA
    (implies (and (inv MT MA)
                (MAETT-p MT)
                (MA-state-p MA)
                (not (b1p (MT-specultv? MT)))
```

```
                (not (b1p (MT-self-modify? MT))))
           (equal (ISA-pc (MT-final-ISA MT)) (MA-PC MA)))
  :hints (("goal" :in-theory (enable weak-inv inv
                                     MT-specultv-p MT-self-modify-p
                                     pc-match-p lift-b-ops)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
           (implies (and (inv MT MA)
                         (MAETT-p MT)
                         (MA-state-p MA)
                         (not (b1p (MT-specultv? MT)))
                         (not (b1p (MT-self-modify? MT))))
                    (equal (MA-PC MA) (ISA-pc (MT-final-ISA MT)))))))
)

(in-theory (disable (:rewrite ISA-pc-MT-final-ISA . 2)))
(theory-invariant (not (and (member-equal '(:rewrite ISA-pc-MT-final-ISA . 1)
                                          theory)
                            (member-equal '(:rewrite ISA-pc-MT-final-ISA . 2)
                                          theory))))

;; A landmark lemma.
;; A newly fetched instruction satisfies INST-inv.
(defthm INST-inv-fetched-inst
    (implies (and (MAETT-p MT)
                  (MA-state-p MA)
                  (inv MT MA)
                  (MA-input-p sigs)
                  (b1p (fetch-inst? ma sigs)))
             (INST-inv (fetched-inst MT (MT-final-ISA MT)
                                     (MT-specultv? MT)
                                     (MT-self-modify? MT))
                       (MA-step MA sigs)))
  :hints (("Goal" :in-theory (e/d (fetched-inst lift-b-ops MA-step-functions
                                                inst-inv-def
                                                MT-def
                                                bv-eqv-iff-equal
                                                INST-function-def)
                                  (exception-relations
                                   incompatible-with-excpt-in-MAETT-lemmas
                                   INST-WB-IF-INST-DATA-ACCS-ERROR-DETECTED
                                   INST-STORE-INST-SYNC-EXCLUSIVE
                                   INST-LSU-IF-INST-STORE
                                   INST-IS-AT-ONE-OF-THE-STAGES)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Proof of INST-inv-step-INST
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; INST-inv-step-INST proves that instruction i preserves instruction
; invariants during the current cycle.
;
; Proof of INST-inv-step-INST contains a number of landmark lemmas.
;   IFU-inst-inv-step-INST
;   DQ-inst-inv-step-INST
;   execute-inst-inv-step-INST
;   complete-inst-inv-step-INST
;   commit-inst-inv-step-INST
; Each lemma confirms that all instruction invariants are preserved
; for i, depending on the stage of i in the next cycle.
;
; Before proceeding on to the proof of each stage dependent lemmas,
```

```
; we first prove lemmas about the jump and exceptions.

(in-theory (enable inst-stg-step-inst))

;; We prove MT-jmp-exintr-before-IFU-DQ-INST-if-flush-all.
;; This lemma suggests that flush-all? is asserted only if
;; there is an instruction which flushes the following instructions.
;; Such instructions are either branch instruction or exception related
;; instructions.

;; The following lemmas are for proving the lemma
;; MT-jmp-exintr-before-IFU-DQ-INST-if-flush-all.
;; The proof sketch is this.  If commit-jmp? is true, there is an
;; instruction stored in ROB at index MT-rob-head.  This instruction at
;; complete stage appears earlier than i in MT, and is an mis-predicted
;; branch instruction which commits this cycle.  The existence of this
;; instruction negates MT-no-jmp-exintr-before.
(local
(defthm uniq-inst-of-tag-if-context-sync
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (or (b1p (commit-jmp? MA))
                      (b1p (leave-excpt? MA))
                      (b1p (enter-excpt? MA))))
             (uniq-inst-of-tag (MT-rob-head MT) MT))
  :hints (("goal" :in-theory (enable commit-jmp? lift-b-ops
                                     enter-excpt? leave-excpt?)))))

(local
(defthm complete-inst-of-tag-if-context-sync
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (or (b1p (commit-jmp? MA))
                      (b1p (leave-excpt? MA))
                      (b1p (enter-excpt? MA))))
             (complete-stg-p (INST-stg (inst-of-tag (MT-rob-head MT) MT))))
  :hints (("goal" :in-theory (e/d (commit-jmp? lift-b-ops
                                               leave-excpt? enter-excpt?
                                               committed-p dispatched-p)
                                  (inst-of-tag-is-dispatched
                                   inst-of-tag-is-not-committed))
                  :use ((:instance inst-of-tag-is-not-committed
                                   (rix (MT-rob-head MT)))
                        (:instance inst-of-tag-is-dispatched
                                   (rix (MT-rob-head MT))))))))

(local
(defthm inst-cause-jmp-inst-of-tag-head
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (or (b1p (commit-jmp? MA))
                      (b1p (leave-excpt? MA))
                      (b1p (enter-excpt? MA))))
             (b1p (inst-cause-jmp? (inst-of-tag (MT-rob-head MT) MT)
                                   MT MA sigs)))
  :hints (("Goal" :in-theory (enable inst-cause-jmp? lift-b-ops)))))

(local
(encapsulate  nil
(local
 (defthm MT-jmp-exintr-before-if-inst-cause-jmp-help
```

```
     (implies (and (distinct-member-p trace)
                   (MAETT-p MT) (MA-state-p MA)
                   (member-in-order i j trace)
                   (member-equal i trace)
                   (member-equal j trace)
                   (b1p (inst-cause-jmp? i MT MA sigs)))
              (not (trace-no-jmp-exintr-before j trace MT MA sigs)))
   :hints (("Goal" :in-theory (enable member-in-order*)))))

; If j follows i in program order, and i causes a jump,
; (MT-no-jmp-exintr-before j MT..) is false.
(defthm MT-jmp-exintr-before-if-inst-cause-jmp
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in-order-p i j MT)
                  (INST-in i MT)
                  (INST-in j MT)
                  (b1p (inst-cause-jmp? i MT MA sigs)))
             (not (MT-no-jmp-exintr-before j MT MA sigs)))
  :hints (("Goal" :in-theory (enable MT-no-jmp-exintr-before
                                     inv MT-distinct-inst-p
                                     weak-inv
                                     INST-in INST-in-order-p)))
  :rule-classes nil)
))

(local
(defthm MT-jmp-exintr-before-IFU-if-context-sync
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (IFU-stg-p (INST-stg i))
                  (or (b1p (commit-jmp? MA))
                      (b1p (leave-excpt? MA))
                      (b1p (enter-excpt? MA))))
             (not (MT-no-jmp-exintr-before i MT MA sigs)))
  :hints (("goal" :use (:instance MT-jmp-exintr-before-if-inst-cause-jmp
                                  (j i)
                                  (i (inst-of-tag (MT-rob-head MT) MT)))))))

(local
(defthm MT-jmp-exintr-before-DQ-if-context-sync
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (DQ-stg-p (INST-stg i))
                  (or (b1p (commit-jmp? MA))
                      (b1p (leave-excpt? MA))
                      (b1p (enter-excpt? MA))))
             (not (MT-no-jmp-exintr-before i MT MA sigs)))
  :hints (("goal" :use (:instance MT-jmp-exintr-before-if-inst-cause-jmp
                                  (j i)
                                  (i (inst-of-tag (MT-rob-head MT) MT)))))))

;; If instruction i is in execute-stg, and a context synchronization occurs
;; in the MA,  (MT-no-jmp-exintr-before i MT ..) is false.
(defthm MT-no-jmp-exintr-before-execute-if-context-sync
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (execute-stg-p (INST-stg i))
                  (or (b1p (commit-jmp? MA))
                      (b1p (leave-excpt? MA))
```

```
                    (b1p (enter-excpt? MA)))
               (MAETT-p MT) (MA-state-p MA) (INST-p i))
          (not (MT-no-jmp-exintr-before i MT MA sigs)))
  :hints (("goal" :use (:instance MT-jmp-exintr-before-if-inst-cause-jmp
                                  (j i)
                                  (i (inst-of-tag (MT-rob-head MT) MT))))
          (use-hint-always
           (:cases ((equal i (INST-OF-TAG (MT-ROB-HEAD MT) MT)))))))

;; This lemma shows that there should be an instruction in MT
;; which causes following instructions to be abandoned, if
;; the MA control line flush-all? is asserted.
;; Suppose i is an instruction at the IFU stage or in the dispatch queue.
;; Flush-all? can be asserted if i is externally interrupted.
;; Otherwise, there should be an instruction in MT which precedes i in
;; program order, and it discards the following instructions including i.
(defthm MT-jmp-exintr-before-IFU-DQ-INST-if-flush-all
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i)
                  (INST-in i MT)
                  (or (IFU-stg-p (INST-stg i))
                      (DQ-stg-p (INST-stg i)))
                  (b1p (flush-all? MA sigs))
                  (not (b1p (INST-exintr-now? i  MA sigs))))
             (not (MT-no-jmp-exintr-before i MT MA sigs)))
  :hints (("goal" :in-theory (enable flush-all? INST-exintr-now?
                                     ex-intr?
                                     lift-b-ops))))

; Suppose i is an instruction in an execution unit. If flush-all? is
; asserted in the MA, branching or other kind of context synchronization is
; caused by a committing instruction which precedes i.
(defthm MT-JMP-EXINTR-BEFORE-execute-INST-IF-FLUSH-ALL
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (execute-stg-p (INST-stg i))
                  (b1p (flush-all? MA sigs))
                  (MAETT-p MT) (MA-state-p MA) (INST-p i) (MA-input-p sigs))
             (not (MT-no-jmp-exintr-before i MT MA sigs)))
  :hints (("goal" :in-theory (enable flush-all? lift-b-ops))))

(encapsulate nil
(local
(defthm leave-excpt-only-if-commit-inst?
    (implies (and (inv MT MA)
                  (not (b1p (commit-inst? MA)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (leave-excpt? MA) 0))
  :hints (("goal" :in-theory (enable leave-excpt? commit-inst? lift-b-ops
                                     equal-b1p-converter)))))

(local
(defthm enter-excpt-only-if-commit-inst?
    (implies (and (inv MT MA)
                  (not (b1p (commit-inst? MA)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (enter-excpt? MA) 0))
  :hints (("goal" :in-theory (enable enter-excpt? commit-inst? lift-b-ops
                                     equal-b1p-converter)))))

; The Instruction i at the head of the ROB retires if i is completed, and
```

```
; one of lines commit-jmp?, leave-excpt? and enter-excpt? is asserted.
(defthm not-complete-step-inst-INST-at-rob-head
    (implies (and (inv MT MA)
                  (MA-state-p MA) (MAETT-p MT)
                  (complete-stg-p (INST-stg (inst-of-tag (MT-rob-head MT) MT)))
                  (or (b1p (commit-jmp? MA))
                      (b1p (leave-excpt? MA))
                      (b1p (enter-excpt? MA))))
             (not (complete-stg-p
                   (INST-stg (step-INST (inst-of-tag (MT-rob-head MT) MT)
                                        MT MA sigs)))))
  :hints (("goal" :in-theory (enable step-INST-opener
                                     step-INST-low-level-functions
                                     INST-commit?
                                     lift-b-ops bv-eqv-iff-equal))))
) ;encapsulate

(encapsulate nil
(local
(defthm consp-MT-non-commit-trace-help-help
    (implies (uniq-inst-of-tag-in-trace rix trace)
             (consp (non-commit-trace trace)))))

(local
(defthm consp-MT-non-commit-trace-help
    (implies (uniq-inst-of-tag (MT-rob-head MT) MT)
             (consp (MT-non-commit-trace MT)))
  :hints (("goal" :in-theory (enable MT-non-commit-trace uniq-inst-of-tag)))))

; If an instruction commits in this cycle, MT-non-commit-trace
; returns non empty list of instructions.
(defthm consp-MT-non-commit-trace
    (implies (and (inv MT MA)
                  (MA-state-p MA) (MAETT-p MT)
                  (b1p (commit-inst? MA)))
             (consp (MT-non-commit-trace MT))))
)

; The first instruction in MT-non-commit-trace has the tag
; (MT-ROB-head MT).
(defthm car-MT-non-commit-trace
    (implies (and (inv MT MA)
                  (MA-state-p MA) (MAETT-p MT)
                  (b1p (commit-inst? MA)))
             (equal (car (MT-non-commit-trace MT))
                    (inst-of-tag (MT-rob-head MT) MT)))
  :hints (("goal" :in-theory
                  (enable (:rewrite car-trace-INST-at-rob-head . 1))
                  :restrict
                  (((:rewrite car-trace-INST-at-rob-head . 1)
                    ((trace (MT-non-commit-trace MT))))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; From here we prove the invariants at each stage.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; The proof of IFU-inst-inv
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; IFU-inst-inv is preserved for instruction i, if i is in IFU-stg in
; the current cycle, and will be in IFU-stg in the next cycle.
(defthm IFU-inst-inv-step-INST-IFU
    (implies (and (IFU-INST-inv I MA)
```

```
                (MAETT-p MT) (MA-state-p MA)
                (INST-p i)
                (not (b1p (flush-all? MA sigs)))
                (IFU-stg-p (INST-stg i))
                (IFU-stg-p (INST-stg (step-INST I MT MA sigs))))
           (IFU-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable IFU-inst-inv
                                     step-INST-low-level-functions
                                     MA-step-functions
                                     lift-b-ops
                                     INST-function-def
                                     step-INST-opener))))

; A landmark lemma
; IFU-inst-inv will hold for instruction i, if it is in IFU-stg in the
; next cycle, given that i is not externally interrupted during this cycle,
; and i is not abandoned by a jump or interrupts by a preceding instruction.
(defthm IFU-inst-inv-step-INST
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (INST-p i)
                (INST-in i MT)
                (INST-inv i MA)
                (not (b1p (INST-exintr-now? i MA sigs)))
                (MT-no-jmp-exintr-before i MT MA sigs)
                (IFU-stg-p (INST-stg (step-INST I MT MA sigs))))
           (IFU-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("Goal" :use ((:instance INST-is-at-one-of-the-stages))
                :in-theory (e/d (INST-inv)
                                (INST-INV-IF-INST-IN)))
          ("Goal'" :cases ((b1p (flush-all? MA sigs))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Proof of DQ-inst-inv
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Lemmas to case split depending on the value of MT-DQ-len.
(defthm MT-DQ-len-0-to-4
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA))
           (or (equal (MT-DQ-len MT) 0) (equal (MT-DQ-len MT) 1)
               (equal (MT-DQ-len MT) 2) (equal (MT-DQ-len MT) 3)
               (equal (MT-DQ-len MT) 4)))
  :hints (("Goal" :in-theory (enable inv misc-inv)))
  :rule-classes nil)

; New-dq-stage returns the dispatch queue stage where a decoded instruction
; is pushed.  It is one of the four stages in the dispatch queue.
(defthm NEW-DQ-stage-one-of-them
    (or (equal (NEW-dq-stage MT MA) '(DQ 0))
        (equal (NEW-dq-stage MT MA) '(DQ 1))
        (equal (NEW-dq-stage MT MA) '(DQ 2))
        (equal (NEW-dq-stage MT MA) '(DQ 3)))
  :hints (("goal" :in-theory (enable NEW-dq-stage)))
  :rule-classes nil)

;;; Following lemmas determines the value of DE-valid? field of each entry
;;; in the dispatching queue.
(defthm DE-valid-DQ-DE0-by-MT-DQ-len
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA))
           (equal (DE-valid? (DQ-DE0 (MA-DQ MA)))
```

```
                  (if (<= (MT-DQ-len MT) 0) 0 1)))
  :hints (("goal" :in-theory (enable inv misc-inv
                                     equal-b1p-converter
                                     correct-entries-in-DQ-p)))
  :rule-classes
  ((:rewrite :corollary
             (implies (and (inv MT MA)
                           (MAETT-p MT) (MA-state-p MA)
                           (<= (MT-DQ-len MT) 0))
                      (equal (DE-valid? (DQ-DE0 (MA-DQ MA))) 0)))
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (MAETT-p MT) (MA-state-p MA)
                           (> (MT-DQ-len MT) 0))
                      (equal (DE-valid? (DQ-DE0 (MA-DQ MA))) 1)))))

(defthm DE-valid-DQ-DE1-by-MT-DQ-len
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (DE-valid? (DQ-DE1 (MA-DQ MA)))
                    (if (<= (MT-DQ-len MT) 1) 0 1)))
  :hints (("goal" :in-theory (enable inv misc-inv
                                     equal-b1p-converter
                                     correct-entries-in-DQ-p)))
  :rule-classes
  ((:rewrite :corollary
             (implies (and (inv MT MA)
                           (MAETT-p MT) (MA-state-p MA)
                           (<= (MT-DQ-len MT) 1))
                      (equal (DE-valid? (DQ-DE1 (MA-DQ MA))) 0)))
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (MAETT-p MT) (MA-state-p MA)
                           (> (MT-DQ-len MT) 1))
                      (equal (DE-valid? (DQ-DE1 (MA-DQ MA))) 1)))))

(defthm DE-valid-DQ-DE2-by-MT-DQ-len
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (DE-valid? (DQ-DE2 (MA-DQ MA)))
                    (if (<= (MT-DQ-len MT) 2) 0 1)))
  :hints (("goal" :in-theory (enable inv misc-inv
                                     equal-b1p-converter
                                     correct-entries-in-DQ-p)))
  :rule-classes
  ((:rewrite :corollary
             (implies (and (inv MT MA)
                           (MAETT-p MT) (MA-state-p MA)
                           (<= (MT-DQ-len MT) 2))
                      (equal (DE-valid? (DQ-DE2 (MA-DQ MA))) 0)))
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (MAETT-p MT) (MA-state-p MA)
                           (> (MT-DQ-len MT) 2))
                      (equal (DE-valid? (DQ-DE2 (MA-DQ MA))) 1)))))

(defthm DE-valid-DQ-DE3-by-MT-DQ-len
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (DE-valid? (DQ-DE3 (MA-DQ MA)))
                    (if (<= (MT-DQ-len MT) 3) 0 1)))
  :hints (("goal" :in-theory (enable inv misc-inv
```

```
                                    equal-b1p-converter
                                    correct-entries-in-DQ-p)))
  :rule-classes
  ((:rewrite :corollary
             (implies (and (inv MT MA)
                           (MAETT-p MT) (MA-state-p MA)
                           (<= (MT-DQ-len MT) 3))
                      (equal (DE-valid? (DQ-DE3 (MA-DQ MA))) 0)))
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (MAETT-p MT) (MA-state-p MA)
                           (> (MT-DQ-len MT) 3))
                      (equal (DE-valid? (DQ-DE3 (MA-DQ MA))) 1)))))

; DQ-full? is true if the dispatch queue length is larger than 3.
(defthm DQ-full-by-MT-DQ-len
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (DQ-full? (MA-DQ MA))
                    (if (<= (MT-DQ-len MT) 3) 0 1)))
  :hints (("goal" :in-theory (enable inv misc-inv
                                    DQ-full?
                                    equal-b1p-converter
                                    correct-entries-in-DQ-p)))
  :rule-classes
  ((:rewrite :corollary
             (implies (and (inv MT MA)
                           (MAETT-p MT) (MA-state-p MA)
                           (<= (MT-DQ-len MT) 3))
                      (equal (DQ-full? (MA-DQ MA)) 0)))
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (MAETT-p MT) (MA-state-p MA)
                           (> (MT-DQ-len MT) 3))
                      (equal (DQ-full? (MA-DQ MA)) 1)))))

; decode-illegal-inst-0 shows that Opcode 0 is not an illegal instruction.
; This is an ad-hoc lemma for the proofs in this book.
(local
(defthm decode-illegal-inst-0
    (equal (decode-illegal-inst? 0 su ra) 0)
  :hints (("goal" :in-theory  (enable decode-illegal-inst?)))))

; IFU-branch-target calculates the branching destination for
; instruction at IFU. INST-br-target calculates the branching
; destination for instruction i.  If an instruction i is at IFU-stg,
; these two functions return the same value.
(defthm IFU-branch-target=INST-br-target
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i)
                  (INST-in i MT)
                  (IFU-stg-p (INST-stg i))
                  (not (INST-fetch-error-detected-p I))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (equal (IFU-branch-target (MA-IFU MA))
                    (INST-br-target i)))
  :hints (("goal" :in-theory (enable IFU-branch-target INST-br-target
                                    INST-pc))))

;; MA-SRF-su-=-INST-su shows the Supervisor/User mode is correct.
```

```
;; Imagine a non-retired instruction i.  Suppose i is not a speculatively
;; executed instruction, and not retired.  The mode in which i should be
;; executed is represented by (INST-su i).  The current mode in MA is
;; determined by  (SRF-su (MA-SRF MA)).  These two values should
;; be identical.

;; A sketch of the proof of MA-SRF-su-=-INST-su is as follows.
;; If i is an instruction which is not retired, and whose specultv? flag
;; is not on, then i appears in MT-non-retire-trace and
;; no partially executed instruction changes the su bit.
;; So the su bits are the same in the pre-ISA states of both i and the first
;; instruction in MT-non-retire-trace.
;;

(local
(encapsulate nil
(local
(defthm MA-su-car-MT-non-retire-trace-=-INST-su-help-induction
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-listp trace)
                  (subtrace-p trace MT)
                  (trace-correct-speculation-p trace)
                  (no-commit-inst-p trace)
                  (member-equal i trace)
                  (not (b1p (inst-specultv? i))))
             (equal (INST-su (car trace)) (INST-su i)))
  :hints (("Goal" :in-theory
                  (enable
                   ISA-before INST-su
                   INST-start-specultv?
                   committed-p
                   lift-b-ops
                   flushed-p
                   INST-exintr-INST-in-if-not-retired
                   inst-specultv-is-not-member-equal-to-trace-all-specultv)
                  :induct t)
          (when-found (SRF-SU (ISA-SRF (INST-PRE-ISA (CAR (CDR TRACE)))))
                      (:cases ((consp (cdr trace))))))
  :rule-classes nil))

(local
(defthm MA-su-car-MT-non-retire-trace-=-INST-su-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-listp trace)
                  (subtrace-p trace MT)
                  (no-commit-inst-p trace)
                  (trace-correct-speculation-p trace)
                  (member-equal i trace)
                  (not (b1p (inst-specultv? i))))
             (equal (SRF-su (ISA-SRF (ISA-before trace MT)))
                    (INST-su i)))
  :hints (("goal" :cases ((endp trace))
                  :in-theory (enable INST-su)
                  :do-not-induct t)
          ("subgoal 2"
           :use (:instance
                 MA-su-car-MT-non-retire-trace-=-INST-su-help-induction)))
  :rule-classes nil))

;  Supervisor/User mode of the ISA state before the first non-committed
```

```
;  instruction in MT is equal to that of i, which is an non-speculative
;  instruction in MT.
(defthm MA-su-car-MT-non-retire-trace-=-INST-su
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (inst-in i MT) (INST-p i)
                  (not (retire-stg-p (INST-stg i)))
                  (not (commit-stg-p (INST-stg i)))
                  (not (b1p (inst-specultv? i))))
             (equal (SRF-su (ISA-SRF (ISA-before (MT-non-commit-trace MT) MT)))
                    (INST-su i)))
  :hints (("goal" :use (:instance MA-su-car-MT-non-retire-trace-=-INST-su-help
                                  (trace (MT-non-commit-trace MT)))))
  :rule-classes nil)
))

;; Supervisor/User mode in the current MA state and the pre-ISA of i
;; are the same.
(defthm MA-SRF-su-=-INST-su
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (inst-in i MT) (INST-p i)
                  (not (retire-stg-p (INST-stg i)))
                  (not (commit-stg-p (INST-stg i)))
                  (not (b1p (inst-specultv? i))))
             (equal (SRF-su (MA-SRF MA))
                    (INST-su i)))
  :hints (("goal" :use (:instance MA-su-car-MT-non-retire-trace-=-INST-su)
                  :in-theory (enable INST-su))))

; DQ0-inst-inv is true for i in the next cycle, if i is in IFU-stg and
; advances to (DQ 0) in this cycle.  Similar lemmas follow.
(defthm DQ0-inst-inv-step-INST-IFU
    (implies (and (inv MT MA)
                  (IFU-INST-inv I MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i)
                  (INST-in i MT)
                  (not (b1p (flush-all? MA sigs)))
                  (IFU-stg-p (INST-stg i))
                  (equal (INST-stg (step-INST I MT MA sigs)) '(DQ 0)))
             (DQ0-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory
                  (e/d (lift-b-ops step-DE0 DE1-out DE2-out DE3-out
                            IFU-INST-inv INST-opcode decode-output INST-su
                            INST-pc INST-ra INST-decode-error?
                            INST-fetch-error-detected-p-iff-INST-fetch-error?
                            INST-decode-error-detected-p-iff-INST-decode-error?
                            exception-relations NEW-dq-stage INST-excpt-flags
                            DQ0-inst-inv)
                       (MT-DQ-len-lemmas))
                  :cases ((b1p (DISPATCH-INST? MA))))
          (use-hint-always (:cases ((b1p (INST-fetch-error? i)))))
          (use-hint-always (:use (:instance MT-DQ-len-0-to-4)))))

(defthm DQ1-inst-inv-step-INST-IFU
    (implies (and (inv MT MA)
                  (IFU-INST-inv I MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i)
                  (INST-in i MT)
                  (not (b1p (flush-all? MA sigs)))
```

```
                (IFU-stg-p (INST-stg i))
                (equal (INST-stg (step-INST I MT MA sigs)) '(DQ 1)))
           (DQ1-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory
                (e/d (lift-b-ops step-DE1
                         DE1-out DE2-out DE3-out IFU-INST-inv
                         INST-opcode decode-output INST-su
                         INST-pc INST-ra exception-relations
                         NEW-dq-stage INST-decode-error?
                         INST-fetch-error-detected-p-iff-INST-fetch-error?
                         INST-decode-error-detected-p-iff-INST-decode-error?
                         INST-excpt-flags DQ1-inst-inv)
                     (MT-DQ-len-lemmas))
                :cases ((b1p (DISPATCH-INST? MA))))
          (use-hint-always (:cases ((b1p (INST-fetch-error? i)))))
          (use-hint-always (:use (:instance MT-DQ-len-0-to-4)))))

(defthm DQ2-inst-inv-step-INST-IFU
    (implies (and (inv MT MA)
                (IFU-INST-inv I MA)
                (MAETT-p MT) (MA-state-p MA)
                (INST-p i)
                (INST-in i MT)
                (not (b1p (flush-all? MA sigs)))
                (IFU-stg-p (INST-stg i))
                (equal (INST-stg (step-INST I MT MA sigs)) '(DQ 2)))
           (DQ2-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory
                (e/d (lift-b-ops step-DE2 DE1-out DE2-out DE3-out
                         IFU-INST-inv INST-opcode decode-output
                         INST-decode-error? INST-su INST-ra INST-pc
                         INST-fetch-error-detected-p-iff-INST-fetch-error?
                         INST-decode-error-detected-p-iff-INST-decode-error?
                         exception-relations NEW-dq-stage INST-excpt-flags
                         DQ2-inst-inv)
                     (MT-DQ-len-lemmas))
                :cases ((b1p (DISPATCH-INST? MA))))
          (use-hint-always (:cases ((b1p (INST-fetch-error? i)))))
          (use-hint-always (:use (:instance MT-DQ-len-0-to-4)))))

(defthm DQ3-inst-inv-step-INST-IFU
    (implies (and (inv MT MA)
                (IFU-INST-inv I MA)
                (MAETT-p MT) (MA-state-p MA)
                (INST-p i)
                (INST-in i MT)
                (not (b1p (flush-all? MA sigs)))
                (IFU-stg-p (INST-stg i))
                (equal (INST-stg (step-INST I MT MA sigs)) '(DQ 3)))
           (DQ3-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory
                (e/d (lift-b-ops step-DE3 DE1-out DE2-out DE3-out
                         IFU-INST-inv INST-opcode INST-ra decode-output
                         INST-fetch-error-detected-p-iff-INST-fetch-error?
                         INST-decode-error-detected-p-iff-INST-decode-error?
                         INST-decode-error? INST-su INST-pc
                         exception-relations NEW-dq-stage INST-excpt-flags
                         DQ3-inst-inv)
                     (MT-DQ-len-lemmas))
                :cases ((b1p (DISPATCH-INST? MA))))
          (use-hint-always (:cases ((b1p (INST-fetch-error? i)))))
          (use-hint-always (:use (:instance MT-DQ-len-0-to-4)))))
```

```
; DQ-inst-inv is true for i, if i's current stage is IFU-stg and
; it advances to DQ-stg.
(defthm DQ-inst-inv-step-INST-IFU
    (implies (and (inv MT MA)
                  (IFU-INST-inv I MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i)
                  (INST-in i MT)
                  (not (b1p (flush-all? MA sigs)))
                  (IFU-stg-p (INST-stg i))
                  (DQ-stg-p (INST-stg (step-INST I MT MA sigs))))
             (DQ-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (e/d (DQ-inst-inv lift-b-ops)
                                  (MT-DQ-len-lemmas))
                  :use (:instance  NEW-DQ-STAGE-ONE-OF-THEM))))

; DQ-inst-inv is true for i if i's current stage if DQ-stg.
(defthm DQ-inst-inv-step-INST-DQ
    (implies (and (inv MT MA)
                  (DQ-INST-inv I MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i)
                  (INST-in i MT)
                  (not (b1p (flush-all? MA sigs)))
                  (DQ-stg-p (INST-stg i))
                  (DQ-stg-p (INST-stg (step-INST I MT MA sigs))))
             (DQ-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory
                  (e/d (DQ-stg-p inst-inv-def step-DE0 step-DE1 step-DE2
                                 step-de3 DE1-out DE2-out DE3-out
                                 lift-b-ops)
                       (MT-DQ-len-lemmas))
                  :cases ((b1p (dispatch-inst? MA))))
          (use-hint-always (:cases ((b1p (INST-fetch-error? i)))))
          (use-hint-always (:use (:instance MT-DQ-len-0-to-4)))))

; A landmark lemma
; DQ-inst-inv is preserved.
(defthm DQ-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (not (b1p (INST-exintr-now? i MA sigs)))
                  (DQ-stg-p (INST-stg (step-INST I MT MA sigs))))
             (DQ-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("Goal" :use ((:instance INST-is-at-one-of-the-stages))
                  :in-theory (e/d (INST-inv)
                                  (INST-INV-IF-INST-IN)))
          ("Goal'" :cases ((b1p (flush-all? MA sigs))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Begin of Proof of execute-inst-inv
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Execute-inst-inv defines the invariant conditions that instructions
; hold in the multi-issue pipelined execution core.
; The proof is organized as follows.
;
;   Lemmas about the stage of dispatched instructions.
```

```
;   Lemmas about dispatch logic
;   Lemmas about register modifiers
;   Lemmas about CDB output
;   Proof of IU-inst-inv-step-INST
;   Proof of MU-inst-inv-step-INST
;   Proof of BU-inst-inv-step-INST
;   Proof of LSU-inst-inv-step-INST
;   Proof of execute-inst-robe-inv-step-INST
; The theorem execute-inst-robe-inv-step-INST guarantees that
; the ROB is recording correct values for instructions.
;;;;;;;;Begin of stage inference lemmas for instructions in DQ ;;;;;;
(deflabel begin-minor-stage-inference-rules)
; A dispatched instruction does not go to stage MU-lch1 directly.
(defthm not-MU-lch1-step-inst-if-DQ-stg
    (implies (DQ-stg-p (INST-stg i))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(MU lch1))))
  :hints (("goal" :in-theory (enable step-INST
                                     STEP-INST-DQ
                                     dispatch-inst))))

; A dispatched instruction does not go to stage MU-lch2 directly.
(defthm not-MU-lch2-step-inst-if-DQ-stg
    (implies (DQ-stg-p (INST-stg i))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(MU lch2))))
  :hints (("goal" :in-theory (enable step-INST
                                     STEP-INST-DQ
                                     dispatch-inst))))

; A dispatched instruction does not go to stage LSU-rbuf directly.
(defthm not-LSU-rbuf-step-inst-if-DQ-stg
    (implies (DQ-stg-p (INST-stg i))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU rbuf))))
  :hints (("goal" :in-theory (enable step-INST
                                     STEP-INST-DQ
                                     dispatch-inst))))

; A dispatched instruction does not go to stage LSU-lch directly.
(defthm not-LSU-lch-step-inst-if-DQ-stg
    (implies (DQ-stg-p (INST-stg i))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU lch))))
  :hints (("goal" :in-theory (enable step-INST
                                     STEP-INST-DQ
                                     dispatch-inst))))

; A dispatched instruction does not go to stage LSU-wbuf0 directly.
(defthm not-LSU-wbuf0-step-inst-if-DQ-stg
    (implies (DQ-stg-p (INST-stg i))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU wbuf0))))
  :hints (("goal" :in-theory (enable step-INST
                                     STEP-INST-DQ
                                     dispatch-inst))))

; A dispatched instruction does not go to stage LSU-wbuf1 directly.
(defthm not-LSU-wbuf1-step-inst-if-DQ-stg
    (implies (DQ-stg-p (INST-stg i))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU wbuf1))))
```

```
  :hints (("goal" :in-theory (enable step-INST
                                     STEP-INST-DQ
                                     dispatch-inst))))

; A dispatched instruction does not go to stage LSU-wbuf0-lch directly.
(defthm not-LSU-wbuf0-lch-step-inst-if-DQ-stg
    (implies (DQ-stg-p (INST-stg i))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU wbuf0 lch))))
  :hints (("goal" :in-theory (enable step-INST
                                     STEP-INST-DQ
                                     dispatch-inst))))

; A dispatched instruction does not go to stage LSU-wbuf1-lch directly.
(defthm not-LSU-wbuf1-lch-step-inst-if-DQ-stg
    (implies (DQ-stg-p (INST-stg i))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU wbuf1 lch))))
  :hints (("goal" :in-theory (enable step-INST
                                     STEP-INST-DQ
                                     dispatch-inst))))

(deflabel end-minor-stage-inference-rules)
(deftheory minor-stage-inference-rules
    (set-difference-theories
     (universal-theory 'end-minor-stage-inference-rules)
     (universal-theory 'begin-minor-stage-inference-rules)))
(in-theory (disable minor-stage-inference-rules))

;;;;;;;;;;;;;End of stage inference lemmas for instructions in DQ ;;;;;;

;;;;;;;;;Lemmas about the dispatch logic.;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Reservation station IU-RS0 is selected for the entry of a new dispatched
; instruction iff IU-RS0 is empty.
(defthm not-select-IU-RS0-if-RS0-valid?
    (equal (select-IU-RS0? (MA-IU MA))
           (b-not (RS-valid? (IU-RS0 (MA-IU MA)))))
  :hints (("goal" :in-theory (enable select-IU-RS0? lift-b-ops
                                     equal-b1p-converter))))

; Reservation station IU-RS1 is not chosen if IU-RS1 is busy.
(defthm not-select-IU-RS1-if-RS1-valid?
    (implies (b1p (RS-valid? (IU-RS1 (MA-IU MA))))
             (equal (select-IU-RS1? (MA-IU MA)) 0))
  :hints (("goal" :in-theory (enable select-IU-RS1? lift-b-ops
                                     dispatch-to-IU?
                                     IU-ready?
                                     equal-b1p-converter))))

; If both IU reservation stations are busy, instructions are not dispatched
; to the IU.
(defthm not-dispatch-to-IU-if-RS-full
    (implies (and (b1p (RS-valid? (IU-RS0 (MA-IU MA))))
                  (b1p (RS-valid? (IU-RS1 (MA-IU MA)))))
             (equal (dispatch-to-IU? MA) 0))
  :hints (("goal" :in-theory (enable lift-b-ops dispatch-to-IU?
                                     DQ-ready-to-IU?
                                     IU-READY?
                                     equal-b1p-converter))))

; A dispatched instruction goes into IU-RS1 if RS1 is the only available
; station.
```

```
(defthm select-IU-RS1-if-not-RS1-valid?
    (implies (and (b1p (RS-valid? (IU-RS0 (MA-IU MA))))
                  (not (b1p (RS-valid? (IU-RS1 (MA-IU MA))))))
             (equal (select-IU-RS1? (MA-IU MA)) 1))
  :hints (("goal" :in-theory (enable lift-b-ops select-IU-RS1?
                                     equal-b1p-converter))))

; Reservation station MU-RS0 is selected over MU-RS1 iff MU-RS0 is empty.
(defthm not-select-MU-RS0-if-RS0-valid?
    (equal (select-MU-RS0? (MA-MU MA))
           (b-not (RS-valid? (MU-RS0 (MA-MU MA)))))
  :hints (("goal" :in-theory (enable select-MU-RS0? lift-b-ops
                                     equal-b1p-converter))))

; Reservation station MU-RS1 is not selected if MU-RS1 is busy.
(defthm not-select-MU-RS1-if-RS1-valid?
    (implies (b1p (RS-valid? (MU-RS1 (MA-MU MA))))
             (equal (select-MU-RS1? (MA-MU MA)) 0))
  :hints (("goal" :in-theory (enable select-MU-RS1? lift-b-ops
                                     dispatch-to-MU?
                                     MU-ready?
                                     equal-b1p-converter))))

; If both MU reservation stations are busy, instructions are not dispatched
; to the MU.
(defthm not-dispatch-to-MU-if-RS-full
    (implies (and (b1p (RS-valid? (MU-RS0 (MA-MU MA))))
                  (b1p (RS-valid? (MU-RS1 (MA-MU MA)))))
             (equal (dispatch-to-MU? MA) 0))
  :hints (("goal" :in-theory (enable lift-b-ops dispatch-to-MU?
                                     DQ-ready-to-MU?
                                     MU-READY?
                                     equal-b1p-converter))))

; A dispatched instruction goes into MU-RS1 if RS1 is the only available
; station.
(defthm select-MU-RS1-if-not-RS1-valid?
    (implies (and (b1p (RS-valid? (MU-RS0 (MA-MU MA))))
                  (not (b1p (RS-valid? (MU-RS1 (MA-MU MA))))))
             (equal (select-MU-RS1? (MA-MU MA)) 1))
  :hints (("goal" :in-theory (enable lift-b-ops select-MU-RS1?
                                     equal-b1p-converter))))

; Reservation station BU-RS0 is selected over BU-RS1 iff BU-RS0 is empty.
(defthm not-select-BU-RS0-if-RS0-valid?
    (equal (select-BU-RS0? (MA-BU MA))
           (b-not (BU-RS-valid? (BU-RS0 (MA-BU MA)))))
  :hints (("goal" :in-theory (enable select-BU-RS0? lift-b-ops
                                     equal-b1p-converter))))

; Reservation station BU-RS1 is not selected if MU-RS1 is busy.
(defthm not-select-BU-RS1-if-RS1-valid?
    (implies (b1p (BU-RS-valid? (BU-RS1 (MA-BU MA))))
             (equal (select-BU-RS1? (MA-BU MA)) 0))
  :hints (("goal" :in-theory (enable select-BU-RS1? lift-b-ops
                                     dispatch-to-BU?
                                     BU-ready?
                                     equal-b1p-converter))))

; If both BU reservation stations are busy, instructions are not dispatched
; to the BU.
(defthm not-dispatch-to-BU-if-RS-full
```

```
    (implies (and (b1p (BU-RS-valid? (BU-RS0 (MA-BU MA))))
                  (b1p (BU-RS-valid? (BU-RS1 (MA-BU MA)))))
             (equal (dispatch-to-BU? MA) 0))
  :hints (("goal" :in-theory (enable lift-b-ops dispatch-to-BU?
                                     DQ-ready-to-BU?
                                     BU-READY?
                                     equal-b1p-converter))))

; A dispatched instruction goes into BU-RS1 if RS1 is the only available
; station.
(defthm select-BU-RS1-if-not-RS1-valid?
    (implies (and (b1p (BU-RS-valid? (BU-RS0 (MA-BU MA))))
                  (not (b1p (BU-RS-valid? (BU-RS1 (MA-BU MA))))))
             (equal (select-BU-RS1? (MA-BU MA)) 1))
  :hints (("goal" :in-theory (enable lift-b-ops select-BU-RS1?
                                     equal-b1p-converter))))

; If both LSU reservation stations are busy, instructions are not dispatched
; to the LSU.
(defthm not-dispatch-to-LSU-if-RS-full
    (implies (and (b1p (LSU-RS-valid? (LSU-RS0 (MA-LSU MA))))
                  (b1p (LSU-RS-valid? (LSU-RS1 (MA-LSU MA)))))
             (equal (dispatch-to-LSU? MA) 0))
  :hints (("goal" :in-theory (enable lift-b-ops dispatch-to-LSU?
                                     DQ-ready-to-LSU?
                                     LSU-READY?
                                     equal-b1p-converter))))

; If LSU-RS0 is the only available reservation station, a dispatched
; instruction goes into LSU-RS0.
(defthm select-LSU-RS0-if-not-RS0-valid?
    (implies (and (MA-state-p MA)
                  (b1p (LSU-RS-valid? (LSU-RS1 (MA-LSU MA))))
                  (not (b1p (LSU-RS-valid? (LSU-RS0 (MA-LSU MA))))))
             (equal (select-LSU-RS0? (MA-LSU MA)) 1))
  :hints (("goal" :in-theory (enable lift-b-ops select-LSU-RS0?
                                     equal-b1p-converter))))

; If LSU-RS1 is the only available reservation station, a dispatched
; instruction goes into LSU-RS1.
(defthm select-LSU-RS1-if-not-RS1-valid?
    (implies (and (MA-state-p MA)
                  (b1p (LSU-RS-valid? (LSU-RS0 (MA-LSU MA))))
                  (not (b1p (LSU-RS-valid? (LSU-RS1 (MA-LSU MA))))))
             (equal (select-LSU-RS1? (MA-LSU MA)) 1))
  :hints (("goal" :in-theory (enable lift-b-ops select-LSU-RS1?
                                     equal-b1p-converter))))

; Select-LSU-RS1 and select-LSU-RS0 are mutually exclusive.
(defthm select-LSU-RS1-select-LSU-RS0
    (implies (MA-state-p MA)
             (equal (select-LSU-RS1? (MA-LSU MA))
                    (b-not (select-LSU-RS0? (MA-LSU MA)))))
  :hints (("goal" :in-theory (enable issue-logic-def LSU-output-def
                                     lift-b-ops))))

; If LSU-RS0 is selected and RS0 is busy, no LSU instruction is dispatched.
(defthm not-dispatch-to-LSU-if-select-LSU-RS0
    (implies (and (MA-state-p MA)
                  (b1p (LSU-RS-valid? (LSU-RS0 (MA-LSU MA))))
                  (b1p (select-LSU-RS0? (MA-LSU MA))))
             (equal (dispatch-to-LSU? MA) 0))
```

```
  :hints (("goal" :in-theory (enable issue-logic-def LSU-output-def
                                     lift-b-ops
                                     equal-b1p-converter dispatch-to-LSU?))))

; If LSU-RS1 is selected and RS1 is busy, no LSU instruction is dispatched.
(defthm not-dispatch-to-LSU-if-select-LSU-RS1
    (implies (and (MA-state-p MA)
                  (b1p (LSU-RS-valid? (LSU-RS1 (MA-LSU MA))))
                  (not (b1p (select-LSU-RS0? (MA-LSU MA)))))
             (equal (dispatch-to-LSU? MA) 0))
  :hints (("goal" :in-theory (enable issue-logic-def LSU-output-def
                                     lift-b-ops
                                     equal-b1p-converter dispatch-to-LSU?))))

; If (INST-no-unit? i) is 0 for instruction i at (DQ 0),
; dispatch-no-unit? is not asserted.
;
; Note: These rules have very simple left-hand sides.  It is recommended
; that these rules are kept local.
(defthm INST-no-unit-if-dispatch-no-unit
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(DQ 0))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (not (b1p (INST-no-unit? i))))
             (equal (dispatch-no-unit? MA) 0))
  :hints (("goal" :in-theory (enable dispatch-no-unit?
                                     equal-b1p-converter
                                     INST-no-unit?
                                     lift-b-ops
                                     exception-relations
                                     inst-excpt-detected-p
                                     DQ-ready-no-unit?)
                  :cases ((INST-fetch-error-detected-p i)))))

; If instruction i at (DQ 0) has caused an exception, dispatch-to-IU? is not
; asserted.  Following three lemmas are similar lemmas for other execution
; units.
(defthm not-dispatch-to-IU-if-excpt-detected
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(DQ 0))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (INST-excpt-detected-p i))
             (equal (dispatch-to-IU? MA) 0))
  :hints (("goal" :in-theory (enable dispatch-to-IU? dispatch-to-LSU?
                                     dispatch-to-BU? dispatch-to-MU?
                                     lift-b-ops
                                     DQ-READY-TO-IU? DQ-READY-TO-BU?
                                     DQ-READY-TO-MU? DQ-READY-TO-LSU?
                                     INST-EXCPT-DETECTED-P))))

(defthm not-dispatch-to-MU-if-excpt-detected
    (implies (and (inv MT MA)
```

```
                (INST-in i MT)
                (equal (INST-stg i) '(DQ 0))
                (MAETT-p MT) (MA-state-p MA)
                (INST-p i)
                (not (b1p (INST-modified? i)))
                (not (b1p (inst-specultv? i)))
                (INST-excpt-detected-p i))
           (equal (dispatch-to-MU? MA) 0))
  :hints (("goal" :in-theory (enable dispatch-to-IU? dispatch-to-LSU?
                                     dispatch-to-BU? dispatch-to-MU?
                                     lift-b-ops
                                     DQ-READY-TO-IU? DQ-READY-TO-BU?
                                     DQ-READY-TO-MU? DQ-READY-TO-LSU?
                                     INST-EXCPT-DETECTED-P))))

(defthm not-dispatch-to-LSU-if-excpt-detected
    (implies (and (inv MT MA)
                (INST-in i MT)
                (equal (INST-stg i) '(DQ 0))
                (MAETT-p MT) (MA-state-p MA)
                (INST-p i)
                (not (b1p (INST-modified? i)))
                (not (b1p (inst-specultv? i)))
                (INST-excpt-detected-p i))
           (equal (dispatch-to-LSU? MA) 0))
  :hints (("goal" :in-theory (enable dispatch-to-IU? dispatch-to-LSU?
                                     dispatch-to-BU? dispatch-to-MU?
                                     lift-b-ops
                                     DQ-READY-TO-IU? DQ-READY-TO-BU?
                                     DQ-READY-TO-MU? DQ-READY-TO-LSU?
                                     INST-EXCPT-DETECTED-P))))

(defthm not-dispatch-to-BU-if-excpt-detected
    (implies (and (inv MT MA)
                (INST-in i MT)
                (equal (INST-stg i) '(DQ 0))
                (MAETT-p MT) (MA-state-p MA)
                (INST-p i)
                (not (b1p (INST-modified? i)))
                (not (b1p (inst-specultv? i)))
                (INST-excpt-detected-p i))
           (equal (dispatch-to-BU? MA) 0))
  :hints (("goal" :in-theory (enable dispatch-to-IU? dispatch-to-LSU?
                                     dispatch-to-BU? dispatch-to-MU?
                                     lift-b-ops
                                     DQ-READY-TO-IU? DQ-READY-TO-BU?
                                     DQ-READY-TO-MU? DQ-READY-TO-LSU?
                                     INST-EXCPT-DETECTED-P))))

; If dispatch-to-IU? is asserted, (INST-IU? i) is true for the instruction
; i at (DQ 0).  The lemma is written as a contrapositive.
; Following three lemmas are similar lemmas for other execution units.
(defthm INST-IU-if-dispatch-to-IU
    (implies (and (inv MT MA)
                (equal (INST-stg i) '(DQ 0))
                (INST-in i MT)
                (MAETT-p MT) (MA-state-p MA)
                (INST-p i)
                (not (b1p (INST-modified? i)))
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-IU? i))))
           (equal (dispatch-to-IU? MA) 0))
```

```
  :hints (("goal" :in-theory (enable dispatch-to-IU?
                                     equal-b1p-converter
                                     INST-IU?
                                     lift-b-ops
                                     exception-relations
                                     DQ-ready-to-IU?)
                  :cases ((INST-fetch-error-detected-p i)))))

(defthm INST-MU-if-dispatch-to-MU
    (implies (and (inv MT MA)
                  (INST-in i MT) (equal (INST-stg i) '(DQ 0))
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-MU? i))))
             (equal (dispatch-to-MU? MA) 0))
  :hints (("goal" :in-theory (enable dispatch-to-MU?
                                     equal-b1p-converter
                                     INST-MU?
                                     lift-b-ops
                                     INST-EXCPT-FLAGS
                                     exception-relations
                                     DQ-ready-to-MU?)
                  :cases ((INST-fetch-error-detected-p i)))))

(defthm INST-BU-if-dispatch-to-BU
    (implies (and (inv MT MA)
                  (INST-in i MT) (equal (INST-stg i) '(DQ 0))
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-BU? i))))
             (equal (dispatch-to-BU? MA) 0))
  :hints (("goal" :in-theory (enable dispatch-to-BU?
                                     equal-b1p-converter
                                     INST-BU?
                                     lift-b-ops
                                     INST-EXCPT-FLAGS
                                     exception-relations
                                     DQ-ready-to-BU?)
                  :cases ((INST-fetch-error-detected-p i)))))

(defthm INST-LSU-if-dispatch-to-LSU
    (implies (and (inv MT MA)
                  (INST-in i MT) (equal (INST-stg i) '(DQ 0))
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-LSU? i))))
             (equal (dispatch-to-LSU? MA) 0))
  :hints (("goal" :in-theory (enable dispatch-to-LSU?
                                     equal-b1p-converter
                                     INST-LSU?
                                     lift-b-ops
                                     INST-EXCPT-FLAGS
                                     exception-relations
                                     DQ-ready-to-LSU?)
                  :cases ((INST-fetch-error-detected-p i)))))
;;;;;;;;;;;; End of Lemmas about dispatch logic. ;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;; Start of Lemmas about Register Modifiers;;;;;;;;;;;;;;;;
; If instruction i has raised an fetch error,  i does not go into the
```

```
; execution stage.   In fact, it advances to the complete stage directly.
(defthm INST-stg-step-INST-if-fetch-error-inst-dispatched
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(DQ 0))
                  (b1p (INST-fetch-error? i))
                  (b1p (dispatch-inst? MA))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (INST-stg (step-INST i MT MA sigs)) '(complete)))
  :hints (("goal" :in-theory (enable dispatch-inst? lift-b-ops
                                     exception-relations))))

;; A load-store instruction i at DQ-stg should not complete in this
;; cycle, because it should advance to the LSU execution stage.
(defthm not-complete-stg-p-step-INST-if-INST-LSU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (DQ-stg-p (INST-stg i))
                  (b1p (INST-LSU? i))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i)))
             (not (complete-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("goal" :in-theory (enable step-INST DQ-stg-p
                                     step-inst-dq
                                     dispatch-inst))))

; A data-access error does not occur before instruction dispatch.
(defthm INST-data-accs-error-detected-p-step-INST-not-dispatched
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p I) (INST-in I MT)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (not (dispatched-p i)))
             (not (INST-data-accs-error-detected-p (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (e/d (INST-data-accs-error-detected-p
                                   INST-LOAD-ACCS-ERROR-DETECTED-P
                                   minor-stage-inference-rules
                                   opcode-inst-type
                                   INST-exunit-relations
                                   INST-DECODE-ERROR-DETECTED-P
                                   INST-STORE-ACCS-ERROR-DETECTED-P)
                                  (INST-STG-STEP-IFU-INST-IF-DQ-FULL
                                   INST-is-at-one-of-the-stages))
                  :use (:instance (:instance INST-is-at-one-of-the-stages)))))

;; No exception occurs in the dispatch queue.
(defthm INST-excpt-detected-p-step-INST-DQ-0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (equal (INST-stg i) '(DQ 0)))
             (equal (INST-excpt-detected-p (step-INST i MT MA sigs))
```

```
                  (INST-excpt-detected-p i)))
  :hints (("goal" :in-theory (enable INST-excpt-detected-p))))

; INST-excpt-flags remains unchanged in the dispatch queue.
(defthm INST-excpt-flags-step-INST-DQ
    (implies (and (inv MT MA)
                  (DQ-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (INST-excpt-flags (step-INST i MT MA sigs))
                    (INST-excpt-flags i)))
  :hints (("goal" :in-theory (enable INST-excpt-flags))))

; Dispatched instruction i is assigned to the tail entry of the ROB.
(defthm INST-rob-step-inst-=-MT-rob-tail
    (implies (and (inv MT MA)
                  (MAETT-p MT)  (MA-state-p MA)
                  (equal (INST-stg i) '(DQ 0))
                  (b1p (dispatch-inst? MA)))
             (equal (INST-tag (STEP-inst i MT MA sigs))
                    (MT-rob-tail MT)))
  :hints (("goal" :in-theory (enable step-INST step-INST-dq
                                     dispatch-inst))))


(encapsulate nil
(local
(defthm read-reg-ISA-before-DQ-DEO-help-induct-2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (rname-p rname)
                  (consp trace)
                  (equal (INST-stg i) '(DQ 0))
                  (member-equal i trace)
                  (no-commit-inst-p trace)
                  (not (trace-exist-LRM-in-ROB-p rname trace)))
             (equal (read-reg rname (ISA-RF (INST-pre-ISA (car trace))))
                    (read-reg rname (ISA-RF (INST-pre-ISA i)))))
  :hints (("goal" :induct (member-equal i trace)
                  :in-theory (enable committed-p))
          (when-found (ISA-RF (INST-PRE-ISA (CAR (CDR TRACE))))
                      (:cases ((consp (cdr trace))))))))

(local
(defthm read-reg-ISA-before-DQ-DEO-help-induct
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (rname-p rname)
                  (consp trace)
                  (equal (INST-stg i) '(DQ 0))
                  (member-equal i trace)
                  (not (trace-exist-LRM-in-ROB-p rname trace)))
             (equal (read-reg rname
                              (trace-RF
                               trace (ISA-RF (INST-pre-ISA (car trace)))))
                    (read-reg rname (ISA-RF (INST-pre-ISA i)))))
```

```
  :hints ((when-found (ISA-RF (INST-PRE-ISA (CAR (CDR TRACE))))
                      (:cases ((consp (cdr trace)))))
          ("goal" :in-theory (enable committed-p)))
  :rule-classes nil))

(local
(defthm read-reg-ISA-before-DQ-DEO-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (rname-p rname)
                  (equal (INST-stg i) '(DQ 0))
                  (not (exist-LRM-in-ROB-p rname MT)))
             (equal (read-reg rname (MT-RF MT))
                    (read-reg rname (ISA-RF (INST-pre-ISA i)))))
  :rule-classes nil
  :hints (("goal" :in-theory (e/d (MT-RF exist-LRM-in-ROB-p
                                         INST-in)
                                  (MT-RF-=-MA-RF))
                  :use (:instance read-reg-ISA-before-DQ-DEO-help-induct
                                  (trace (MT-trace MT)))))))

; An important lemma.
; Suppose instruction i is at (DQ 0), and there is no register modifier
; in ROB, then the actual register contains the correct operand for i.
(defthm read-reg-ISA-before-DQ-DEO
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (rname-p rname)
                  (equal (INST-stg i) '(DQ 0))
                  (not (exist-LRM-in-ROB-p rname MT)))
             (equal (read-reg rname (ISA-RF (INST-pre-ISA i)))
                    (read-reg rname (MA-RF MA))))
  :hints (("goal" :use (:instance read-reg-ISA-before-DQ-DEO-help))))
)

(encapsulate nil
(local
(defthm read-sreg-ISA-before-DQ-DEO-help-induct-2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (srname-p rname)
                  (consp trace)
                  (equal (INST-stg i) '(DQ 0))
                  (member-equal i trace)
                  (no-commit-inst-p trace)
                  (not (b1p (trace-specultv-at-dispatch? trace)))
                  (not (trace-exist-LSRM-in-ROB-p rname trace)))
             (equal (read-sreg rname (ISA-SRF (INST-pre-ISA (car trace))))
                    (read-sreg rname (ISA-SRF (INST-pre-ISA i)))))
  :hints (("goal" :induct (member-equal i trace)
                  :in-theory (enable committed-p lift-b-ops
                                     INST-exintr-INST-in-if-not-retired
                                     INST-start-specultv?))
          (when-found (ISA-SRF (INST-PRE-ISA (CAR (CDR TRACE))))
                      (:cases ((consp (cdr trace))))))))

(local
(defthm read-sreg-ISA-before-DQ-DEO-help-induct
```

```
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (srname-p rname)
                  (consp trace)
                  (equal (INST-stg i) '(DQ 0))
                  (member-equal i trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (trace-exist-LSRM-in-ROB-p rname trace))
                  (not (b1p (trace-specultv-at-dispatch? trace))))
             (equal (read-sreg rname
                               (trace-SRF
                                trace (ISA-SRF (INST-pre-ISA (car trace)))))
                    (read-sreg rname (ISA-SRF (INST-pre-ISA i)))))
  :hints ((when-found (ISA-SRF (INST-PRE-ISA (CAR (CDR TRACE))))
                      (:cases ((consp (cdr trace)))))
          ("goal" :in-theory (enable committed-p)))
  :rule-classes nil))

(local
(defthm read-sreg-ISA-before-DQ-DE0-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT)
                  (srname-p rname)
                  (equal (INST-stg i) '(DQ 0))
                  (not (exist-LSRM-in-ROB-p rname MT))
                  (not (b1p (MT-specultv-at-dispatch? MT))))
             (equal (read-sreg rname (MT-SRF MT))
                    (read-sreg rname (ISA-SRF (INST-pre-ISA i)))))
  :rule-classes nil
  :hints (("goal" :in-theory (e/d (MT-SRF exist-LSRM-in-ROB-p
                                   MT-specultv-at-dispatch? INST-in)
                                  (MT-SRF-=-MA-SRF))
                  :use (:instance read-sreg-ISA-before-DQ-DE0-help-induct
                                  (trace (MT-trace MT)))))))

; An important lemma.
; Suppose instruction i is at (DQ 0).  If no register modifier of
; special register rname is in ROB, the actual special register rname
; contains the correct register value for i.
(defthm read-sreg-ISA-before-DQ-DE0
     (implies (and (inv MT MA)
                   (equal (INST-stg i) '(DQ 0))
                   (MAETT-p MT) (MA-state-p MA)
                   (INST-in i MT)
                   (srname-p rname)
                   (not (exist-LSRM-in-ROB-p rname MT))
                   (not (b1p (MT-specultv-at-dispatch? MT))))
              (equal (read-sreg rname (ISA-SRF (INST-pre-ISA i)))
                     (read-sreg rname (MA-SRF MA))))
  :hints (("goal" :use (:instance read-sreg-ISA-before-DQ-DE0-help))))
)

; Instruction i is MTSR or MFSR instruction, and RA does not
; designate a legitimate special register, i is an illegal instruction.
(defthm INST-decode-error-if-INST-ra-not-srname-p
    (implies (and (or (equal (INST-opcode i) 9)
                      (equal (INST-opcode i) 10))
                  (not (srname-p (INST-RA I)))
                  (inv MT MA)
```

```
                (MAETT-p MT) (MA-state-p MA)
                (INST-p i) (INST-in i MT))
           (equal (INST-decode-error? i) 1))
  :hints (("goal" :in-theory (enable INST-decode-error?
                                     equal-b1p-converter
                                     decode-illegal-inst?
                                     srname-p rname-p
                                     INST-opcode
                                     lift-b-ops))))

; Instruction i is at (DQ 0), and output DQ-read-val1 from the dispatch
; logic is the first source operand value for instruction i.
(defthm DQ-read-val1-INST-src-val1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(DQ 0))
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (b1p (DQ-out-ready1? (MA-DQ MA))))
             (equal (DQ-read-val1 (MA-DQ MA) MA) (INST-src-val1 i)))
  :hints (("goal" :in-theory
                  (e/d (DQ-out-ready1?
                        DQ-read-val1
                        INST-SRC-val1
                        INST-cntlv decode rdb logbit*
                        INST-modified-at-dispatch-off-if-undispatched-inst-in
                        equal-b1p-converter
                        lift-b-ops)
                       (INST-decode-error-if-INST-ra-not-srname-p)))
          (when-found (EQUAL '9 (INST-OPCODE I))
                      (:use
                       (:instance INST-decode-error-if-INST-ra-not-srname-p)))
          (when-found (b1p (INST-decode-error? i))
                      (:cases ((b1p (INST-fetch-error? i)))))))

; Instruction i is at (DQ 0), and output DQ-read-val2 from a dispatch
; logic is the second source operand value of instruction i.
(defthm DQ-read-val2-INST-src-val2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(DQ 0))
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (b1p (DQ-out-ready2? (MA-DQ MA))))
             (equal (read-reg (DQ-out-reg2 (MA-DQ MA)) (MA-RF MA))
                    (INST-src-val2 i)))
  :hints (("goal" :in-theory
                  (e/d (DQ-out-ready2?
                        DQ-out-reg2
                        INST-SRC-val2
                        INST-cntlv decode rdb logbit*
                        equal-b1p-converter
                        lift-b-ops)
                       (INST-decode-error-if-INST-ra-not-srname-p)))))

; Instruction i is at (DQ 0), and output DQ-read-val3 from a dispatch
```

```
; logic is the third source operand value of instruction i.
(defthm DQ-read-val3-INST-src-val3
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(DQ 0))
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (b1p (DQ-out-ready3? (MA-DQ MA))))
             (equal (read-reg (DQ-out-reg3 (MA-DQ MA)) (MA-RF MA))
                    (INST-src-val3 i)))
  :hints (("goal" :in-theory
                  (e/d (DQ-out-ready3?
                        DQ-out-reg3
                        INST-SRC-val3
                        INST-cntlv decode rdb logbit*
                        equal-b1p-converter
                        lift-b-ops)
                       (INST-decode-error-if-INST-ra-not-srname-p)))))

(in-theory (enable IFU-is-last-inst DQ0-is-earlier-than-other-DQ))

(local
(defthm INST-dest-val-read-reg-INST-post-ISA
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-excpt? I)))
                  (not (b1p (INST-exintr? i)))
                  (reg-modifier-p rname i))
             (equal (INST-dest-val i)
                    (read-reg rname (ISA-RF (INST-post-ISA i)))))
  :hints (("goal" :in-theory (enable INST-dest-val
                                     reg-modifier-p
                                     INST-function-def
                                     DECODE-ILLEGAL-INST?
                                     lift-b-ops
                                     ISA-step ISA-step-functions)))))

(local
(defthm INST-dest-val-read-sreg-INST-post-ISA
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-excpt? I)))
                  (not (b1p (INST-exintr? i)))
                  (sreg-modifier-p rname i))
             (equal (INST-dest-val i)
                    (read-sreg rname (ISA-SRF (INST-post-ISA i)))))
  :hints (("goal" :in-theory (enable INST-dest-val
                                     sreg-modifier-p
                                     INST-function-def
                                     DECODE-ILLEGAL-INST?
                                     lift-b-ops
                                     ISA-step ISA-step-functions)))))
```

```
; A local lemma.
; Suppose instruction k precedes i in program order.
; Suppose s0 and s1 are the pre-ISA state of k and i, respectively.
; If there is no modifier of register rname between k and i,
; the value of register rname in s0 and s1 are the same.
; Note: This rule has a very bad form.  The LHS term pattern matches the
; RHS.  It is possible that this rule causes an infinite loop.  So
; we disable the rule.
(local
(defthm read-reg-unchanged-if-not-trace-exist-LRM-before-p
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (not (b1p (inst-specultv? I)))
                  (not (b1p (inst-modified? I)))
                  (not (trace-exist-LRM-before-p i rname trace))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-listp trace)
                  (INST-p i) (rname-p rname)
                  (member-equal i trace))
             (equal (read-reg rname (ISA-RF (INST-pre-ISA i)))
                    (read-reg rname (ISA-RF (INST-pre-ISA (car trace))))))
  :hints ((when-found (ISA-RF (INST-PRE-ISA (CAR (CDR TRACE))))
                      (:cases ((consp (cdr trace))))))))

(local
(defthm read-sreg-unchanged-if-not-trace-exist-LSRM-before-p
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (not (b1p (inst-specultv? I)))
                  (not (b1p (inst-modified? I)))
                  (not (trace-exist-LSRM-before-p i rname trace))
                  (no-commit-inst-p trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-listp trace)
                  (INST-p i) (srname-p rname)
                  (member-equal i trace))
             (equal (read-sreg rname (ISA-SRF (INST-pre-ISA i)))
                    (read-sreg rname (ISA-SRF (INST-pre-ISA (car trace))))))
  :hints ((when-found (ISA-SRF (INST-PRE-ISA (CAR (CDR TRACE))))
                      (:cases ((consp (cdr trace)))))
          ("goal" :in-theory (enable INST-exintr-INST-in-if-not-retired
                                     committed-p)))))

(local
(in-theory
 (disable read-reg-unchanged-if-not-trace-exist-LRM-before-p
          read-sreg-unchanged-if-not-trace-exist-LSRM-before-p)))

(encapsulate nil
(local
 (defthm INST-dest-val-LRM-before-help-help
     (implies (and (consp trace)
                   (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (subtrace-p trace MT)
                   (rname-p rname)
                   (INST-listp trace)
                   (not (b1p (inst-specultv? I)))
                   (not (b1p (inst-modified? I)))
                   (member-equal i (cdr trace))
                   (reg-modifier-p rname (car trace))
                   (not (committed-p (car trace)))
```

```
                  (not (trace-exist-LRM-before-p i rname
                                                 (cdr trace))))
              (equal (read-reg rname (ISA-RF (INST-pre-ISA i)))
                     (INST-dest-val (car trace))))
   :Hints (("goal" :cases ((consp (cdr trace)))
                   :in-theory (enable
                     INST-exintr-INST-in-if-not-retired
                     read-reg-unchanged-if-not-trace-exist-LRM-before-p
                     committed-p)
                   :restrict
                   ((read-reg-unchanged-if-not-trace-exist-LRM-before-p
                     ((trace (cdr trace)))))))))

(local
 (defthm INST-dest-val-LRM-before-help
     (implies (and (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (subtrace-p trace MT)
                   (INST-listp trace)
                   (rname-p rname)
                   (not (b1p (inst-specultv? I)))
                   (not (b1p (inst-modified? I)))
                   (INST-p i) (member-equal i trace)
                   (trace-exist-uncommitted-LRM-before-p i rname trace))
     (equal (INST-dest-val (trace-LRM-before i rname trace))
            (read-reg rname (ISA-RF (INST-pre-ISA i)))))
   :hints (("goal" :restrict
                   ((INST-dest-val-LRM-before-help-help
                     ((trace trace))))))))

; An important lemma.
; Suppose j is a last modifier of register rname before i.  And s is
; the pre-ISA i.  Then the destination value of j is the same as the
; value of register rname in s.
(defthm INST-dest-val-LRM-before*
    (implies (and (inv MT MA)
                  (exist-uncommitted-LRM-before-p i rname MT)
                  (not (b1p (inst-specultv? I)))
                  (not (b1p (inst-modified? I)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (rname-p rname))
             (equal (INST-dest-val (LRM-before i rname MT))
                    (read-reg rname (ISA-RF (INST-pre-ISA i)))))
  :hints (("goal" :in-theory (enable LRM-before
                                     exist-uncommitted-LRM-before-p
                                     INST-in))))
)

;; Another presentation of the same original theorem
(defthm INST-dest-val-LRM-before
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? I)))
                  (not (b1p (inst-modified? I)))
                  (rname-p rname)
                  (exist-LRM-before-p i rname MT)
                  (not (committed-p (LRM-before i rname MT))))
             (equal (INST-dest-val (LRM-before i rname MT))
                    (read-reg rname (ISA-RF (INST-pre-ISA i)))))
  :hints (("goal" :in-theory
           (enable exist-LRM-before-p-and-exist-uncommitted-LRM-before-p)))
```

```
  :rule-classes nil)

(encapsulate nil
(local
(defthm INST-dest-val-LSRM-before-help-help
    (implies (and (consp trace)
                  (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (srname-p rname)
                  (INST-listp trace)
                  (not (b1p (inst-specultv? I)))
                  (not (b1p (inst-modified? I)))
                  (member-equal i (cdr trace))
                  (sreg-modifier-p rname (car trace))
                  (not (committed-p (car trace)))
                  (not (trace-exist-uncommitted-LSRM-before-p i rname (cdr trace))))
             (equal (read-sreg rname (ISA-SRF (INST-pre-ISA i)))
                    (INST-dest-val (car trace))))
  :hints (("goal"
           :cases ((consp (cdr trace)))
           :in-theory
           (enable
            read-sreg-unchanged-if-not-trace-exist-LSRM-before-p
            INST-exintr-INST-in-if-not-retired
            committed-p)
           :restrict
           ((read-sreg-unchanged-if-not-trace-exist-LSRM-before-p
             ((trace (cdr trace)))))))))

(local
(defthm INST-dest-val-LSRM-before-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (srname-p rname)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (INST-p i) (member-equal i trace)
                  (trace-exist-uncommitted-LSRM-before-p i rname trace))
     (equal (INST-dest-val (trace-LSRM-before i rname trace))
            (read-sreg rname (ISA-SRF (INST-pre-ISA i)))))
  :hints (("goal" :restrict
                  ((INST-dest-val-LSRM-before-help-help
                    ((trace trace)))))))

; Suppose j is a the last modifier of special register rname before i.
; The destination value of j is the value of the special register rname
; in the pre-ISA state of i.
(defthm INST-dest-val-LSRM-before*
    (implies (and (inv MT MA)
                  (exist-uncommitted-LSRM-before-p i rname MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (srname-p rname))
             (equal (INST-dest-val (LSRM-before i rname MT))
                    (read-sreg rname (ISA-SRF (INST-pre-ISA i)))))
  :hints (("goal" :in-theory (enable LSRM-before
                                     exist-uncommitted-LSRM-before-p
```

```
                                    INST-in))))
)

; Another presentation of INST-dest-val-LSRM-before.
(defthm INST-dest-val-LSRM-before
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT) (srname-p rname)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (exist-LSRM-before-p i rname MT)
                  (not (committed-p (LSRM-before i rname MT))))
             (equal (INST-dest-val (LSRM-before i rname MT))
                    (read-sreg rname (ISA-SRF (INST-pre-ISA i)))))
  :hints (("goal" :in-theory
           (enable exist-LSRM-before-p-and-exist-uncommitted-LSRM-before-p)))
  :rule-classes nil)

; If j is the last register modifier before i, and i is not speculatively
; executed instruction, then j is not speculatively executed, either.
(defthm inst-specultv-LRM-before
    (implies (and (inv MT MA)
                  (exist-LRM-before-p i rname MT)
                  (not (b1p (inst-specultv? I)))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (inst-specultv? (LRM-before i rname MT))
                    0))
  :hints (("goal" :in-theory (e/d (equal-b1p-converter)
                                  (INST-in-order-p-inst-specultv))
                  :use (:instance INST-in-order-p-inst-specultv
                                  (i (LRM-before i rname MT))
                                  (j i)))))

; see inst-specultv-LRM-before
(defthm inst-specultv-LSRM-before
    (implies (and (inv MT MA)
                  (exist-LSRM-before-p i rname MT)
                  (not (b1p (inst-specultv? I)))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (inst-specultv? (LSRM-before i rname MT))
                    0))
  :hints (("goal" :in-theory (e/d (equal-b1p-converter)
                                  (INST-in-order-p-inst-specultv))
                  :use (:instance INST-in-order-p-inst-specultv
                                  (i (LSRM-before i rname MT))
                                  (j i)))))

; If j is the last register modifier before i, and i's modified flag is not
; on, then j's modified flag is not on either.
(defthm INST-modified-LRM-before
    (implies (and (inv MT MA)
                  (exist-LRM-before-p i rname MT)
                  (not (b1p (INST-modified? I)))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (INST-modified? (LRM-before i rname MT))
                    0))
  :hints (("goal" :in-theory (e/d (equal-b1p-converter)
```

```
                                (INST-in-order-p-INST-modified))
                  :use (:instance INST-in-order-p-INST-modified
                                (i (LRM-before i rname MT))
                                (j i)))))

; See INST-modified-LRM-before.
(defthm INST-modified-LSRM-before
    (implies (and (inv MT MA)
                  (exist-LSRM-before-p i rname MT)
                  (not (b1p (INST-modified? I)))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (equal (INST-modified? (LSRM-before i rname MT))
                    0))
  :hints (("goal" :in-theory (e/d (equal-b1p-converter)
                                (INST-in-order-p-INST-modified))
                  :use (:instance INST-in-order-p-INST-modified
                                (i (LSRM-before i rname MT))
                                (j i)))))

; If j is the last register modifier before i, and i is not speculatively
; executed, then no fetch error is detected for j.
(defthm INST-fetch-error-detected-p-LRM-before
    (implies (and (inv MT MA)
                  (exist-uncommitted-LRM-before-p i rname MT)
                  (not (b1p (inst-specultv? I)))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (not (INST-fetch-error-detected-p
                   (LRM-before i rname MT))))
  :hints (("goal" :in-theory (e/d (equal-b1p-converter
                                   INST-start-specultv?
                                   INST-excpt?
                                   lift-b-ops)
                                (INST-in-order-p-INST-start-specultv))
                  :use (:instance INST-in-order-p-INST-start-specultv
                                (i (LRM-before i rname MT))
                                (j i)))))

; See INST-fetch-error-detected-p-LRM-before.
(defthm INST-fetch-error-detected-p-LSRM-before
    (implies (and (inv MT MA)
                  (exist-uncommitted-LSRM-before-p i rname MT)
                  (not (b1p (inst-specultv? I)))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (not (INST-fetch-error-detected-p
                   (LSRM-before i rname MT))))
  :hints (("goal" :in-theory (e/d (equal-b1p-converter
                                      INST-start-specultv?
                                      INST-excpt?
                                      lift-b-ops)
                                (INST-in-order-p-INST-start-specultv))
                  :use (:instance INST-in-order-p-INST-start-specultv
                                (i (LSRM-before i rname MT))
                                (j i)))))

; If j is the last register modifier before i in program order, and
; i is not speculatively executed, then j has not raised an exception.
```

```
(defthm INST-excpt-detected-p-LRM-before
    (implies (and (inv MT MA)
                  (exist-uncommitted-LRM-before-p i rname MT)
                  (not (b1p (inst-specultv? I)))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (not (INST-excpt-detected-p
                   (LRM-before i rname MT))))
    :hints (("goal" :in-theory (e/d (equal-b1p-converter
                                     INST-start-specultv?
                                     INST-excpt?
                                     lift-b-ops)
                                    (INST-in-order-p-INST-start-specultv))
                  :use (:instance INST-in-order-p-INST-start-specultv
                                  (i (LRM-before i rname MT))
                                  (j i))))))

; See INST-excpt-detected-p-LRM-before
(defthm INST-excpt-detected-p-LSRM-before
    (implies (and (inv MT MA)
                  (exist-uncommitted-LSRM-before-p i rname MT)
                  (not (b1p (inst-specultv? I)))
                  (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i))
             (not (INST-excpt-detected-p
                   (LSRM-before i rname MT))))
  :hints (("goal" :in-theory (e/d (equal-b1p-converter
                                     INST-start-specultv?
                                     INST-excpt?
                                     lift-b-ops)
                                    (INST-in-order-p-INST-start-specultv))
                  :use (:instance INST-in-order-p-INST-start-specultv
                                  (i (LSRM-before i rname MT))
                                  (j i))))))
;;;;;;;;;;;;;;;End of last register modifier theory ;;;;;;;;;;;;;;;;;;

;;;;;;;;Lemmas about CDB output;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  When IU-RS0-issue-ready? is on, there is an instruction at IU RS0.
(defthm uniq-inst-of-tag-if-IU-RS0-issue-ready
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (IU-RS0-issue-ready? (MA-IU MA))))
             (uniq-inst-of-tag (RS-tag (IU-RS0 (MA-IU MA))) MT))
  :hints (("goal" :in-theory (enable lift-b-ops IU-RS0-issue-ready?
                                     IU-RS-field-INST-at-lemmas))))

;  When IU-RS1-issue-ready? is on, there is an instruction at IU RS1.
(defthm uniq-inst-of-tag-if-IU-RS1-issue-ready
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (IU-RS1-issue-ready? (MA-IU MA))))
             (uniq-inst-of-tag (RS-tag (IU-RS1 (MA-IU MA))) MT))
  :hints (("goal" :in-theory (enable lift-b-ops IU-RS1-issue-ready?
                                     IU-RS-field-INST-at-lemmas))))

; When CDB-for-IU? is on, an instruction is assigned to the ROB entry
; designated by CDB-tag.
(defthm uniq-inst-of-tag-if-CDB-for-IU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
```

```
                (b1p (CDB-for-IU? MA)))
           (uniq-inst-of-tag (CDB-tag MA) MT))
  :hints (("goal" :in-theory (enable CDB-tag lift-b-ops CDB-for-IU?
                                     IU-output-tag)
                  :cases ((b1p (IU-RS0-ISSUE-READY? (MA-IU MA)))))))

; When CDB-for-MU? is on, an instruction is assigned to the ROB entry
; designated by CDB-tag.
(defthm uniq-inst-of-tag-if-CDB-for-MU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (CDB-for-MU? MA)))
             (uniq-inst-of-tag (CDB-tag MA) MT))
  :hints (("goal" :in-theory (enable CDB-def lift-b-ops
                                     MU-field-lemmas))))

; When CDB-for-BU? is on, an instruction is assigned to the ROB entry
; designated by CDB-tag.
(defthm uniq-inst-of-tag-if-CDB-for-BU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (CDB-for-BU? MA)))
             (uniq-inst-of-tag (CDB-tag MA) MT))
  :hints (("goal" :in-theory (enable CDB-def lift-b-ops BU-output-def))))

; When CDB-for-LSU? is on, an instruction is assigned to the ROB entry
; designated by CDB-tag.
(defthm uniq-inst-of-tag-if-CDB-for-LSU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (CDB-for-LSU? MA)))
             (uniq-inst-of-tag (CDB-tag MA) MT))
  :hints (("goal" :in-theory (enable CDB-def lift-b-ops LSU-output-def
                                     LSU-field-INST-at-lemmas))))

; When CDB-ready-for? is 1 with the Tomasulo's tag rix, an instruction is
; assigned to the ROB entry designated by rix.
(defthm uniq-inst-of-tag-if-CDB-ready-for
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (rob-index-p rix)
                  (b1p (CDB-ready-for? rix MA)))
             (uniq-inst-of-tag rix MT))
  :hints (("goal" :in-theory (enable lift-b-ops CDB-ready-for?
                                     bv-eqv-iff-equal)
                  :use (:instance cases-of-CDB-ready))))

; If CDB-ready-for is 1 with Tomasulo's tag rix, and an IU instruction
; is assigned to the ROB entry designated by rix, CDB-for-IU? is 1.
(defthm CDB-for-IU-if-CDB-ready-for-INST-IU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (CDB-ready-for? rix MA))
                  (b1p (INST-IU? (inst-of-tag rix MT)))
                  (rob-index-p rix)
                  (not (b1p (inst-specultv? (inst-of-tag rix MT))))
                  (not (b1p (INST-modified? (inst-of-tag rix MT)))))
             (b1p (CDB-for-IU? MA)))
  :hints (("goal" :in-theory (enable CDB-def
                                     BU-output-def
                                     bv-eqv-iff-equal
                                     BU-RS-field-inst-at-lemmas
```

```
                                    MU-field-inst-at-lemmas
                                    LSU-field-inst-at-lemmas
                                    CDB-ready?
                                    lift-b-ops
                                    equal-b1p-converter)))
  :rule-classes nil)

; If CDB-ready-for is 1 with Tomasulo's tag rix, and an BU instruction
; is assigned to the ROB entry designated by rix, CDB-for-BU? is 1.
(defthm CDB-for-BU-if-CDB-ready-for-INST-BU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (CDB-ready-for? rix MA))
                  (b1p (INST-BU? (inst-of-tag rix MT)))
                  (rob-index-p rix)
                  (not (b1p (inst-specultv? (inst-of-tag rix MT))))
                  (not (b1p (INST-modified? (inst-of-tag rix MT)))))
             (b1p (CDB-for-BU? MA)))
  :hints (("goal" :in-theory (enable CDB-ready-for?
                                    CDB-for-BU? CDB-tag
                                    CDB-for-IU?
                                    CDB-for-IU?
                                    CDB-for-MU?
                                    CDB-READY?
                                    CDB-for-LSU?
                                    IU-output-tag
                                    BU-output-tag
                                    BU-RS-field-inst-at-lemmas
                                    MU-field-inst-at-lemmas
                                    LSU-field-inst-at-lemmas
                                    IU-RS0-ISSUE-READY?
                                    IU-RS1-ISSUE-READY?
                                    inst-type-exclusive-2
                                    bv-eqv-iff-equal
                                    lift-b-ops
                                    equal-b1p-converter)))
  :rule-classes nil)

; If CDB-ready-for is 1 with Tomasulo's tag rix, and an LSU instruction
; is assigned to the ROB entry designated by rix, CDB-for-LSU? is 1.
(defthm CDB-for-LSU-if-CDB-ready-for-INST-LSU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (CDB-ready-for? rix MA))
                  (b1p (INST-LSU? (inst-of-tag rix MT)))
                  (rob-index-p rix)
                  (not (b1p (inst-specultv? (inst-of-tag rix MT))))
                  (not (b1p (INST-modified? (inst-of-tag rix MT)))))
             (b1p (CDB-for-LSU? MA)))
  :hints (("goal" :in-theory (enable CDB-def
                                    BU-output-def
                                    IU-output-def
                                    BU-RS-field-inst-at-lemmas
                                    MU-field-lemmas
                                    LSU-field-lemmas
                                    LSU-output-def
                                    bv-eqv-iff-equal
                                    lift-b-ops
                                    equal-b1p-converter)))
  :rule-classes nil)

; If CDB-ready-for is 1 with Tomasulo's tag rix, and an MU instruction
```

```
; is assigned to the ROB entry designated by rix, CDB-for-MU? is 1.
(defthm CDB-for-MU-if-CDB-ready-for-INST-MU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (CDB-ready-for? rix MA))
                  (b1p (INST-MU? (inst-of-tag rix MT)))
                  (rob-index-p rix)
                  (not (b1p (inst-specultv? (inst-of-tag rix MT))))
                  (not (b1p (INST-modified? (inst-of-tag rix MT)))))
             (b1p (CDB-for-MU? MA)))
  :hints (("goal" :in-theory (enable CDB-def
                                     BU-output-def
                                     IU-output-def
                                     LSU-output-def
                                     BU-RS-field-inst-at-lemmas
                                     MU-field-inst-at-lemmas
                                     LSU-field-inst-at-lemmas
                                     MU-output-def
                                     bv-eqv-iff-equal
                                     lift-b-ops
                                     equal-b1p-converter)))
  :rule-classes nil)

; If the operand of the completing instruction is 0(ADD), signal
; IU-output-val has destination value for an add instruction.
; A help lemma to prove CDB-val-inst-dest-val-inst-of-tag-opcode-0.
(local
(defthm IU-output-val-INST-add-dest-val
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-opcode
                          (inst-of-tag (IU-output-tag (MA-IU MA)) MT))
                         0)
                  (b1p (CDB-for-IU? MA))
                  (not (b1p (inst-specultv?
                             (inst-of-tag (IU-output-tag (MA-IU MA)) MT))))
                  (not (b1p (INST-modified?
                             (inst-of-tag (IU-output-tag (MA-IU MA)) MT)))))
             (equal (IU-output-val (MA-IU MA))
                    (INST-add-dest-val (inst-of-tag
                                        (IU-output-tag (MA-IU MA)) MT))))
  :hints (("goal" :in-theory (enable CDB-for-IU? lift-b-ops
                                     IU-output-tag IU-output-val
                                     IU-RS0-ISSUE-READY?
                                     IU-RS1-ISSUE-READY?
                                     IU-RS-field-INST-at-lemmas
                                     INST-IU-op?
                                     INST-cntlv
                                     INST-add-dest-val)))))

; If the opcode of the completing instruction is 9,
; the correct destination value is on the CDB.
(defthm CDB-val-inst-dest-val-inst-of-tag-opcode-0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (CDB-ready-for? rix MA))
                  (rob-index-p rix)
                  (equal (INST-opcode (inst-of-tag rix MT)) 0)
                  (not (b1p (inst-specultv? (inst-of-tag rix MT))))
                  (not (b1p (INST-modified? (inst-of-tag rix MT)))))
             (equal (CDB-val MA)
                    (INST-dest-val (inst-of-tag rix MT)))))
```

```
  :hints (("goal" :in-theory (enable CDB-val INST-dest-val
                                     CDB-tag
                                     lift-b-ops
                                     INST-opcode
                                     CDB-ready-for?
                                     opcode-inst-type)
                  :use (:instance CDB-for-IU-if-CDB-ready-for-INST-IU))))

(local
(defthm IU-output-val-INST-mfsr-dest-val
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-opcode
                          (inst-of-tag (IU-output-tag (MA-IU MA)) MT))
                         9)
                  (b1p (CDB-for-IU? MA))
                  (not (b1p (inst-specultv?
                             (inst-of-tag (IU-output-tag (MA-IU MA)) MT))))
                  (not (b1p (INST-modified?
                             (inst-of-tag (IU-output-tag (MA-IU MA)) MT)))))
             (equal (IU-output-val (MA-IU MA))
                    (INST-mfsr-dest-val (inst-of-tag
                                         (IU-output-tag (MA-IU MA)) MT))))
  :hints (("goal" :in-theory (enable CDB-for-IU? lift-b-ops
                                     IU-output-tag IU-output-val
                                     IU-RS0-ISSUE-READY?
                                     IU-RS1-ISSUE-READY?
                                     IU-RS-field-INST-at-lemmas
                                     INST-IU-op?
                                     INST-cntlv
                                     INST-mfsr-dest-val)))))

; If the opcode of the completing instruction is 9,
; the correct destination value is on the CDB.
(defthm CDB-val-inst-dest-val-inst-of-tag-opcode-9
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (rob-index-p rix)
                  (b1p (CDB-ready-for? rix MA))
                  (equal (INST-opcode (inst-of-tag rix MT)) 9)
                  (not (b1p (inst-specultv? (inst-of-tag rix MT))))
                  (not (b1p (INST-modified? (inst-of-tag rix MT)))))
             (equal (CDB-val MA)
                    (INST-dest-val (inst-of-tag rix MT))))
    :hints (("goal" :in-theory (enable CDB-val INST-dest-val
                                       CDB-tag
                                       lift-b-ops
                                       INST-opcode
                                       INST-IU? INST-cntlv
                                       CDB-ready-for?
                                       opcode-inst-type)
                    :use (:instance CDB-for-IU-if-CDB-ready-for-INST-IU))))

(local
(defthm IU-output-val-INST-mtsr-dest-val
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-opcode
                          (inst-of-tag (IU-output-tag (MA-IU MA)) MT))
                         10)
                  (b1p (CDB-for-IU? MA))
                  (not (b1p (inst-specultv?
                             (inst-of-tag (IU-output-tag (MA-IU MA)) MT))))
```

```
                (not (b1p (INST-modified?
                           (inst-of-tag (IU-output-tag (MA-IU MA)) MT)))))
           (equal (IU-output-val (MA-IU MA))
                  (INST-mtsr-dest-val (inst-of-tag
                                       (IU-output-tag (MA-IU MA)) MT))))
  :hints (("goal" :in-theory (enable CDB-for-IU? lift-b-ops
                                     IU-output-tag IU-output-val
                                     IU-RS0-ISSUE-READY?
                                     IU-RS1-ISSUE-READY?
                                     IU-RS-field-INST-at-lemmas
                                     INST-IU-op?
                                     INST-cntlv
                                     INST-mtsr-dest-val)))))

; If the opcode of the completing instruction is 10,
; the correct destination value is on the CDB.
(defthm CDB-val-inst-dest-val-inst-of-tag-opcode-10
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (rob-index-p rix)
                  (b1p (CDB-ready-for? rix MA))
                  (equal (INST-opcode (inst-of-tag rix MT)) 10)
                  (not (b1p (inst-specultv? (inst-of-tag rix MT))))
                  (not (b1p (INST-modified? (inst-of-tag rix MT)))))
             (equal (CDB-val MA)
                    (INST-dest-val (inst-of-tag rix MT))))
  :hints (("goal" :in-theory (enable CDB-val INST-dest-val
                                     CDB-tag
                                     lift-b-ops
                                     INST-opcode
                                     INST-IU? INST-cntlv
                                     CDB-ready-for?
                                     opcode-inst-type)
                  :use (:instance CDB-for-IU-if-CDB-ready-for-INST-IU))))

(encapsulate nil
(local
(defthm LSU-latch-val-INST-ld-dest-val-help
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg stg MT)
                  (or (equal stg '(LSU wbuf0 lch))
                      (equal stg '(LSU wbuf1 lch)))
                  (not (b1p (inst-specultv? (INST-at-stg stg MT))))
                  (not (b1p (INST-modified? (INST-at-stg stg MT))))
                  (MAETT-p MT) (MA-state-p MA))
             (not (equal (opcode (INST-word (INST-at-stg stg MT))) 3)))
  :hints (("goal" :cases ((b1p (INST-store? (INST-at-stg stg MT))))
                  :in-theory (enable LSU-field-lemmas))
          ("subgoal 1" :in-theory (e/d (opcode-inst-type INST-opcode)
                                       (LSU-STORE-IF-AT-LSU-WBUF))))))
(local
(defthm LSU-latch-val-INST-ld-dest-val
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (LSU-latch-valid? (LSU-lch (MA-LSU MA))))
                  (equal (INST-opcode (INST-at-stgs '((LSU lch)
                                                      (LSU wbuf0 lch)
                                                      (LSU wbuf1 lch))
                                                    MT))
                         3)
                  (not (INST-load-accs-error-detected-p
                        (INST-at-stgs '((LSU lch)
                                        (LSU wbuf0 lch)
```

```
                                        (LSU wbuf1 lch))
                                      MT)))
                (not (b1p (inst-specultv?
                           (INST-at-stgs '((LSU lch)
                                        (LSU wbuf0 lch)
                                        (LSU wbuf1 lch))
                                      MT))))
                (not (b1p (INST-modified?
                           (INST-at-stgs '((LSU lch)
                                        (LSU wbuf0 lch)
                                        (LSU wbuf1 lch))
                                      MT)))))
          (equal (LSU-latch-val (LSU-lch (MA-LSU MA)))
                 (INST-ld-dest-val (INST-at-stgs '((LSU lch)
                                                   (LSU wbuf0 lch)
                                                   (LSU wbuf1 lch))
                                                 MT))))
  :hints (("goal" :use ((:instance uniq-inst-at-stgs*
                                   (stgs '((LSU lch)
                                           (LSU wbuf0 lch)
                                           (LSU wbuf1 lch))))
                        (:instance uniq-inst-at-stgs*
                                   (stgs '((LSU wbuf0 lch)
                                           (LSU wbuf1 lch)))))
                :in-theory (e/d (inst-at-stgs* INST-DEST-VAL
                                  LSU-field-inst-at-lemmas
                                  INST-opcode INST-LD-DEST-VAL)
                                ()))))

; If the opcode of the completing instruction is 3,
; the correct destination value is on the CDB.
(defthm CDB-val-inst-dest-val-inst-of-tag-opcode-3
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (rob-index-p rix)
                  (b1p (CDB-ready-for? rix MA))
                  (equal (INST-opcode (inst-of-tag rix MT)) 3)
                  (not (INST-load-accs-error-detected-p (inst-of-tag rix MT)))
                  (not (b1p (inst-specultv? (inst-of-tag rix MT))))
                  (not (b1p (INST-modified? (inst-of-tag rix MT)))))
             (equal (CDB-val MA)
                    (INST-dest-val (inst-of-tag rix MT))))
  :hints (("goal" :in-theory (enable CDB-val CDB-ready-for?
                                     INST-dest-val opcode-inst-type
                                     CDB-tag
                                     INST-LSU-if-INST-store
                                     INST-LSU-if-INST-load
                                     LSU-field-inst-at-lemmas
                                     CDB-FOR-LSU?
                                     INST-opcode
                                     lift-b-ops)
                  :use (:instance CDB-for-LSU-if-CDB-ready-for-INST-LSU))))
)

(encapsulate nil
(local
(defthm LSU-latch-val-INST-ld-im-dest-val-help
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg stg MT)
                  (or (equal stg '(LSU wbuf0 lch))
                      (equal stg '(LSU wbuf1 lch)))
                  (not (b1p (inst-specultv? (INST-at-stg stg MT))))
                  (not (b1p (INST-modified? (INST-at-stg stg MT))))
```

```
                (MAETT-p MT) (MA-state-p MA))
           (not (equal (opcode (INST-word (INST-at-stg stg MT))) 6)))
  :hints (("goal" :cases ((b1p (INST-store? (INST-at-stg stg MT))))
                  :in-theory (enable LSU-field-lemmas))
          ("subgoal 1" :in-theory (e/d (opcode-inst-type INST-opcode)
                                       (LSU-STORE-IF-AT-LSU-WBUF))))))
(local
(defthm LSU-latch-val-INST-ld-im-dest-val
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (LSU-latch-valid? (LSU-lch (MA-LSU MA))))
                  (equal (INST-opcode (INST-at-stgs '((LSU lch)
                                                      (LSU wbuf0 lch)
                                                      (LSU wbuf1 lch))
                                                    MT))
                         6)
                  (not (INST-load-accs-error-detected-p
                        (INST-at-stgs '((LSU lch)
                                        (LSU wbuf0 lch)
                                        (LSU wbuf1 lch))
                                      MT)))
                  (not (b1p (inst-specultv?
                             (INST-at-stgs '((LSU lch)
                                             (LSU wbuf0 lch)
                                             (LSU wbuf1 lch))
                                           MT))))
                  (not (b1p (INST-modified?
                             (INST-at-stgs '((LSU lch)
                                             (LSU wbuf0 lch)
                                             (LSU wbuf1 lch))
                                           MT)))))
     (equal (LSU-latch-val (LSU-lch (MA-LSU MA)))
            (INST-ld-im-dest-val (INST-at-stgs '((LSU lch)
                                                 (LSU wbuf0 lch)
                                                 (LSU wbuf1 lch))
                                               MT))))
  :hints (("goal" :use ((:instance uniq-inst-at-stgs*
                                   (stgs '((LSU lch)
                                           (LSU wbuf0 lch)
                                           (LSU wbuf1 lch))))
                        (:instance uniq-inst-at-stgs*
                                   (stgs '((LSU wbuf0 lch)
                                           (LSU wbuf1 lch)))))
                  :in-theory (e/d (inst-at-stgs* INST-DEST-VAL
                                                 LSU-field-inst-at-lemmas
                                   INST-opcode INST-LD-DEST-VAL)
                                  ()))))))

; If the opcode of the completing instruction is 6,
; the correct destination value is on the CDB.
(defthm CDB-val-inst-dest-val-inst-of-tag-opcode-6
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (rob-index-p rix)
                  (b1p (CDB-ready-for? rix MA))
                  (equal (INST-opcode (inst-of-tag rix MT)) 6)
                  (not (INST-load-accs-error-detected-p (inst-of-tag rix MT)))
                  (not (b1p (inst-specultv? (inst-of-tag rix MT))))
                  (not (b1p (INST-modified? (inst-of-tag rix MT)))))
             (equal (CDB-val MA)
                    (INST-dest-val (inst-of-tag rix MT))))
    :hints (("goal" :in-theory (enable CDB-val CDB-ready-for?
                                       INST-dest-val opcode-inst-type
```

```
                                    CDB-tag
                                    INST-LSU-if-INST-store
                                    INST-LSU-if-INST-load
                                    CDB-FOR-LSU?
                                    LSU-field-inst-at-lemmas
                                    INST-opcode
                                    lift-b-ops)
                  :use (:instance CDB-for-LSU-if-CDB-ready-for-INST-LSU))))
)

; If the opcode of the completing instruction is 1,
; the correct destination value is on the CDB.
(defthm CDB-val-inst-dest-val-inst-of-tag-opcode-1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (rob-index-p rix)
                  (b1p (CDB-ready-for? rix MA))
                  (equal (INST-opcode (inst-of-tag rix MT)) 1)
                  (not (b1p (inst-specultv? (inst-of-tag rix MT))))
                  (not (b1p (INST-modified? (inst-of-tag rix MT)))))
             (equal (CDB-val MA)
                    (INST-dest-val (inst-of-tag rix MT))))
  :hints (("goal" :in-theory (enable CDB-val CDB-ready-for?
                                    INST-dest-val opcode-inst-type
                                    CDB-tag
                                    CDB-FOR-MU?
                                    MU-RS-field-inst-at-lemmas
                                    INST-opcode
                                    lift-b-ops
                                    MU-field-lemmas
                                    INST-MULT-dest-val)
                  :use (:instance CDB-for-MU-if-CDB-ready-for-INST-MU))))

; The val signal on the CDB contains the correct result of the
; instruction completing in this cycle.
(defthm CDB-val-inst-dest-val
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (CDB-ready-for? rix MA))
                  (rob-index-p rix)
                  (INST-writeback-p (inst-of-tag rix MT))
                  (not (INST-excpt-detected-p (inst-of-tag rix MT)))
                  (not (b1p (inst-specultv? (inst-of-tag rix MT))))
                  (not (b1p (INST-modified? (inst-of-tag rix MT)))))
             (equal (CDB-val MA)
                    (INST-dest-val (inst-of-tag rix MT))))
  :hints (("goal" :in-theory (enable INST-writeback-p INST-opcode
                                    INST-DATA-ACCS-ERROR-DETECTED-P
                                    INST-excpt-detected-p))))

(defthm CDB-val-inst-dest-val*
    (let ((j (inst-of-tag (CDB-tag MA) MT)))
      (implies (and (and (inv MT MA) (MAETT-p MT) (MA-state-p MA) )
                    (INST-writeback-p j)
                    (not (b1p (inst-specultv? j)))
                    (not (b1p (INST-modified? j)))
                    (not (INST-excpt-detected-p j))
                    (b1p (CDB-ready? MA)))
               (equal (CDB-val MA) (INST-dest-val j))))
  :hints (("goal" :in-theory (enable lift-b-ops CDB-ready-for?
                                    bv-eqv-iff-equal )
                  :restrict ((CDB-val-inst-dest-val
                              ((rix (CDB-tag MA)))))))))
```

```
  :rule-classes nil)

(in-theory (disable CDB-val-inst-dest-val-inst-of-tag-opcode-0
                    CDB-val-inst-dest-val-inst-of-tag-opcode-1
                    CDB-val-inst-dest-val-inst-of-tag-opcode-3
                    CDB-val-inst-dest-val-inst-of-tag-opcode-6
                    CDB-val-inst-dest-val-inst-of-tag-opcode-9
                    CDB-val-inst-dest-val-inst-of-tag-opcode-10))
(in-theory (disable CDB-val-inst-dest-val))

(in-theory (disable uniq-inst-of-tag-if-CDB-ready-for))

;;;;;;;;End of lemmas about CDB output;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;Proof of INST-inv continues;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;Proof of IU-inst-inv;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(encapsulate nil
(local
(defthm complete-stg-LRM-before
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (exist-uncommitted-LRM-before-p I rname MT)
                  (equal (INST-stg i) '(DQ 0))
                  (b1p (robe-complete?
                        (nth-robe
                         (INST-tag (LRM-before i rname MT))
                         (MA-rob MA)))))
     (complete-stg-p (INST-stg (LRM-before i rname MT))))
  :hints (("goal" :in-theory (disable INST-in-order-LRM-before
                                      INST-is-at-one-of-the-stages
                                      committed-p)
                  :use ((:instance INST-is-at-one-of-the-stages
                                   (i (LRM-before i rname MT)))
                        (:instance INST-in-order-LRM-before))))))

(local
(defthm complete-stg-LSRM-before
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (exist-uncommitted-LSRM-before-p I rname MT)
                  (equal (INST-stg i) '(DQ 0))
                  (b1p (robe-complete?
                        (nth-robe
                         (INST-tag (LSRM-before i rname MT))
                         (MA-rob MA)))))
     (complete-stg-p (INST-stg (LSRM-before i rname MT))))
  :hints (("goal" :in-theory (disable INST-in-order-LSRM-before
                                      INST-is-at-one-of-the-stages
                                      committed-p)
                  :use ((:instance INST-is-at-one-of-the-stages
                                   (i (LSRM-before i rname MT)))
                        (:instance INST-in-order-LSRM-before)))))
)

; A critical lemma.
; When an instruction is dispatched, its operand may be obtained from
; the ROB, where register values are temporarily stored.  Signal
; DQ-out-tag1 from the dispatch logic designates the ROB entry where
; the first operand for the dispatched instruction is temporarily
; stored.  The robe-val field of the ROB entry designated by
; DQ-out-tag1 is equal to the ideal first operand value
```

```
; (INST-src-val1 i) for instruction i, when the instruction at the
; ROB entry has completed its execution.
(defthm robe-val-DQ-out-tag1-INST-src-val1-if-IU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-IU? MA))
                  (equal (INST-stg i) '(DQ 0))
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (b1p (robe-complete? (nth-robe (DQ-out-tag1 (MA-DQ MA))
                                                 (MA-ROB MA))))
                  (not (b1p (DQ-out-ready1? (MA-DQ MA)))))
             (equal (robe-val (nth-robe (DQ-out-tag1 (MA-DQ MA))
                                        (MA-ROB MA)))
                    (INST-src-val1 i)))
  :hints (("goal" :in-theory (e/d (DQ-out-ready1?
                                   DQ-out-tag1
                                   INST-SRC-val1
                                   INST-cntlv
                                   INST-DECODE-ERROR-DETECTED-P
                                   dispatch-to-IU?
                                   DQ-ready-to-IU?
                                   decode logbit* rdb
                                   exception-relations
                                   INST-exunit-relations
                                   equal-b1p-converter
                                   lift-b-ops)
                                  (INST-decode-error-if-INST-ra-not-srname-p
                                   RETIRED-DISPATCHED-INST
                                   UNIQ-INST-OF-TAG-IF-ROBE-VALID
                                   ROBE-COMPLETE-NTH-ROBE-RIX
                                   COMPLETED-DISPATCHED-INST
                                   )))
          (when-found (EQUAL '9 (INST-OPCODE I))
                      (:use
                       (:instance INST-decode-error-if-INST-ra-not-srname-p)))
          (when-found (b1p (INST-decode-error? i))
                      (:cases ((b1p (INST-fetch-error? i)))))))


; A critical lemma.
; As discussed in the comment for the previous lemma, operands may be
; obtained from the ROB.  The robe-val field of the ROB entry designated by
; DQ-out-tag2 is equal to the ideal operand value (INST-src-val2 i) for the
; dispatched instruction i, if the instruction in the ROB entry has completed.
(defthm robe-val-DQ-out-tag2-INST-src-val2-if-IU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-IU? MA))
                  (equal (INST-stg i) '(DQ 0))
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (b1p (robe-complete? (nth-robe (DQ-out-tag2 (MA-DQ MA))
                                                 (MA-ROB MA))))
                  (not (b1p (DQ-out-ready2? (MA-DQ MA)))))
             (equal (robe-val (nth-robe (DQ-out-tag2 (MA-DQ MA))
```

```
                                        (MA-ROB MA)))
                    (INST-src-val2 i)))
  :hints (("goal" :in-theory (e/d (DQ-out-ready2?
                                   DQ-out-tag2
                                   INST-SRC-val2
                                   INST-cntlv
                                   INST-DECODE-ERROR-DETECTED-P
                                   dispatch-to-IU?
                                   DQ-ready-to-IU?
                                   equal-b1p-converter
                                   lift-b-ops)
                                  (INST-decode-error-if-INST-ra-not-srname-p
                                   RETIRED-DISPATCHED-INST
                                   UNIQ-INST-OF-TAG-IF-ROBE-VALID
                                   ROBE-COMPLETE-NTH-ROBE-RIX
                                   COMPLETED-DISPATCHED-INST)))))
)

; A critical lemma.
; CDB-val-INST-src-val1-if-dispatch-IU
; The value on the CDB is the first operand for instruction i, when
; CDB-ready-for? is true for DQ-out-tag1, i.e., the
; first source operand is just in time ready on the CDB.
; This shows that the forwarding logic is working fine.
;
; Proof of CDB-val-INST-src-val1-if-dispatch-IU
; The sketch of the proof is like this.
; (INST-src-val1 i) = (read-reg rname (ISA-RF (INST-Pre-ISA i))).
; The right hand side is actually the destination value of
; the last register modifier before i
;                   = (INST-dest-val (LRM-before i rname MT)
; Since i is at stage (DQ 0),
;                   = (INST-dest-val (inst-of-tag (reg-ref-tag rname MA))
;                   = (INST-dest-val (inst-of-tag (DQ-out-tag1 (MA-DQ-MA))))
; This value is returned through CDB when CDB-ready-for? flag is on,
;                   = (CDB-val MA)
(defthm CDB-val-INST-src-val1-if-dispatch-IU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-IU? MA))
                  (equal (INST-stg i) '(DQ 0))
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (b1p (CDB-ready-for? (DQ-out-tag1 (MA-DQ MA)) MA))
                  (not (b1p (robe-complete? (nth-robe (DQ-out-tag1 (MA-DQ MA))
                                                      (MA-ROB MA)))))
                  (not (b1p (DQ-out-ready1? (MA-DQ MA)))))
             (equal (CDB-val MA) (INST-src-val1 i)))
  :hints (("goal" :in-theory (enable INST-src-val1
                                     CDB-val-inst-dest-val
                                     DQ-out-tag1
                                     lift-b-ops
                                     INST-cntlv
                                     DISPATCH-TO-IU?
                                     DQ-READY-TO-IU?
                                     INST-DECODE-ERROR-DETECTED-P
                                     DQ-out-ready1?))))

; Similar to CDB-val-INST-src-val1-if-dispatch-IU.
```

```
; The value on the CDB is the second operand of for instruction i, when
; CDB-ready-for? is true for DQ-out-tag2.
; See CDB-val-INST-src-val1-if-dispatch-IU.
(defthm CDB-val-INST-src-val2-if-dispatch-IU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-IU? MA))
                  (equal (INST-stg i) '(DQ 0))
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (b1p (CDB-ready-for? (DQ-out-tag2 (MA-DQ MA)) MA))
                  (not (b1p (robe-complete? (nth-robe (DQ-out-tag2 (MA-DQ MA))
                                                      (MA-ROB MA)))))
                  (not (b1p (DQ-out-ready2? (MA-DQ MA)))))
             (equal (CDB-val MA) (INST-src-val2 i)))
  :hints (("goal" :in-theory (enable INST-src-val2
                                     CDB-val-inst-dest-val
                                     DQ-out-tag2
                                     lift-b-ops
                                     INST-cntlv
                                     DISPATCH-TO-IU?
                                     DQ-READY-TO-IU?
                                     INST-DECODE-ERROR-DETECTED-P
                                     DQ-out-ready2?))))

(in-theory (disable CDB-val-INST-src-val1-if-dispatch-IU
                    CDB-val-INST-src-val2-if-dispatch-IU))

; An output signal dispatch-val1 from the dispatching logic designates
; the first operand of the dispatched instruction i.
; The proof derives from
;  robe-val-DQ-out-tag1-INST-src-val1-if-IU and
;  CDB-val-INST-src-val1-if-dispatch-IU.
(defthm dispatch-val1-INST-src-val1-if-dispatched-to-IU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (b1p (dispatch-to-IU? MA))
                  (b1p (dispatch-ready1? MA)))
             (equal (dispatch-val1 MA) (INST-src-val1 i)))
  :hints (("goal" :in-theory (enable dispatch-ready1? dispatch-val1
                                     CDB-val-INST-src-val1-if-dispatch-IU
                                     lift-b-ops))))

; An output signal dispatch-val2 from the dispatching logic is the second
; operand of the dispatched instruction i.
(defthm dispatch-val2-INST-src-val2-if-dispatch-to-IU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(DQ 0))
                  (INST-p i) (INST-in i MT)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (not (INST-fetch-error-detected-p i))
```

```
                (not (INST-decode-error-detected-p i))
                (b1p (dispatch-to-IU? MA))
                (b1p (dispatch-ready2? MA)))
           (equal (dispatch-val2 MA) (INST-src-val2 i)))
  :hints (("goal" :in-theory (enable dispatch-ready2? dispatch-val2
                                     CDB-val-INST-src-val2-if-dispatch-IU
                                     lift-b-ops))))

; The instruction invariants are preserved when instruction moves from
; (DQ 0) to (IU RS0).
(defthm IU-RS0-inst-inv-step-INST-from-DQ0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                  (equal (INST-stg i) '(DQ 0))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(IU RS0)))
             (IU-RS0-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (e/d (lift-b-ops inst-inv-def
                                     step-IU
                                     step-IU-RS0
                                     INST-IU-op?
                                     INST-cntlv
                                     equal-b1p-converter
                                     INST-EXCPT-DETECTED-P
                                     exception-relations
                                     decode logbit* rdb
                                     dispatch-inst?
                                     dispatch-cntlv)
                                   (INST-decode-error-if-INST-ra-not-srname-p))
                  :cases ((b1p (dispatch-inst? MA))))
          ("subgoal 1" :cases ((b1p (INST-fetch-error? I))))
          ("subgoal 1.2" :cases ((b1p (INST-decode-error? I))))
          ("subgoal 1.2.2"
           :use (:instance INST-decode-error-if-INST-ra-not-srname-p))))

; The instruction invariants are preserved when instruction moves from
; (DQ 0) to (IU RS1).
(defthm IU-RS1-inst-inv-step-INST-from-DQ0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                  (equal (INST-stg i) '(DQ 0))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(IU RS1)))
             (IU-RS1-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable lift-b-ops inst-inv-def
                                     step-IU
                                     step-IU-RS1
                                     INST-IU-op?
                                     INST-cntlv
                                     equal-b1p-converter
                                     exception-relations
                                     decode logbit* rdb
```

```
                                  dispatch-inst?)
                  :cases ((b1p (dispatch-inst? MA))))
          ("subgoal 1" :cases ((b1p (INST-fetch-error? I))))
          ("subgoal 1.2" :cases ((b1p (INST-decode-error? I))))
          ("subgoal 1.2.2"
           :use (:instance INST-decode-error-if-INST-ra-not-srname-p))))

; The RA field of an integer unit instruction that manipulates a special
; register designates a legitimate special register.
(defthm srname-p-INST-ra-cntlv-operand3
    (implies (and (b1p (logbit 3 (cntlv-operand (INST-cntlv i))))
                  (IU-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (srname-p (INST-ra i)))
  :hints (("goal" :in-theory (e/d (inst-inv-def equal-b1p-converter
                                                IU-stg-p)
                                  (INST-inv-if-INST-in))
                  :use (:instance INST-inv-if-INST-in))))

; No exception occurs in the IU.
(defthm INST-excpt-detected-p-step-INST-IU
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (IU-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA) (INST-p i)
                  (MA-input-p sigs))
             (not (INST-excpt-detected-p (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (e/d (INST-excpt-detected-p IU-STG-P) ()))))

; Suppose instruction i is in a reservation station.  If the first
; operand for i is not ready in the reservation station, there must be
; a register modifier of the operand register.  Since operand type is
; 1, it is the value of the register specified by ra.
(defthm exist-uncommitted-LRM-INST-ra-if-IU-RS0-cntlv-operand0
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(IU RS0))
                  (not (b1p (RS-ready1? (IU-RS0 (MA-IU MA)))))
                  (b1p (logbit 0 (cntlv-operand (INST-cntlv i))))
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (INST-p i) (INST-in i MT))
             (exist-uncommitted-LRM-before-p i (INST-ra i) MT))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Suppose instruction i is in a reservation station.
; If the first operand for i is not ready in the reservation station,
; there must be a register modifier of the operand register.
; Since operand type is 2, it is the value of the register specified by rc.
(defthm exist-uncommitted-LRM-INST-ra-if-IU-RS0-cntlv-operand2
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(IU RS0))
                  (not (b1p (RS-ready1? (IU-RS0 (MA-IU MA)))))
                  (b1p (logbit 2 (cntlv-operand (INST-cntlv i))))
                  (not (b1p (inst-specultv? i)))
```

```
                (not (b1p (INST-modified? i)))
                (MAETT-p MT) (MA-state-p MA)
                (INST-p i) (INST-in i MT))
           (exist-uncommitted-LRM-before-p i (INST-rc i) MT))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Suppose instruction i is in a reservation station.  If the first
; operand for i is not ready in the reservation station, there must be
; a register modifier of the operand register.  Since operand type is
; 3, it is the value of the register specified by ra.
(defthm exist-uncommitted-LRM-INST-ra-if-IU-RS0-cntlv-operand3
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(IU RS0))
                  (not (b1p (RS-ready1? (IU-RS0 (MA-IU MA)))))
                  (b1p (logbit 3 (cntlv-operand (INST-cntlv i))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (exist-uncommitted-LSRM-before-p i (INST-ra i) MT))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Suppose instruction i is in a reservation station.
; If the second operand for i is not ready in the reservation station,
; there must be a register modifier of the operand register.
(defthm exist-uncommitted-LRM-INST-rb-if-IU-RS0-cntlv-operand0
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(IU RS0))
                  (not (b1p (RS-ready2? (IU-RS0 (MA-IU MA)))))
                  (b1p (logbit 0 (cntlv-operand (INST-cntlv i))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (exist-uncommitted-LRM-before-p i (INST-rb i) MT))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Src1 field of a reservation station contains the tag to refer to
; the last register modifier of the first operand register.
; This lemma directly derives from the consistent-RS-p.
; See the def. of consistent-RS-p.  Similar to the following
; two lemmas.
(defthm IU-RS0-src1-INST-tag-LRM-before-if-cntlv-operand0
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(IU RS0))
                  (b1p (logbit 0 (cntlv-operand (INST-cntlv i))))
                  (not (b1p (RS-ready1? (IU-RS0 (MA-IU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (RS-src1 (IU-RS0 (MA-IU MA)))
                    (INST-tag (LRM-before i (INST-ra i) MT))))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

(defthm IU-RS0-src1-INST-tag-LRM-before-if-cntlv-operand2
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(IU RS0))
```

```
                (b1p (logbit 2 (cntlv-operand (INST-cntlv i))))
                (not (b1p (RS-ready1? (IU-RS0 (MA-IU MA)))))
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (MAETT-p MT) (MA-state-p MA)
                (INST-p i) (INST-in i MT))
           (equal (RS-src1 (IU-RS0 (MA-IU MA)))
                  (INST-tag (LRM-before i (INST-rc i) MT))))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

(defthm IU-RS0-src1-INST-tag-LRM-before-if-cntlv-operand3
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(IU RS0))
                  (b1p (logbit 3 (cntlv-operand (INST-cntlv i))))
                  (not (b1p (RS-ready1? (IU-RS0 (MA-IU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (RS-src1 (IU-RS0 (MA-IU MA)))
                    (INST-tag (LSRM-before i (INST-ra i) MT))))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Src2 field of a reservation station contains the tag to refer to
; the last register modifier of the second operand register.
; This lemma directly derives from the consistent-RS-p.
(defthm IU-RS0-src2-INST-tag-LRM-before
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(IU RS0))
                  (b1p (logbit 0 (cntlv-operand (INST-cntlv i))))
                  (not (b1p (RS-ready2? (IU-RS0 (MA-IU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (RS-src2 (IU-RS0 (MA-IU MA)))
                    (INST-tag (LRM-before i (INST-rb i) MT))))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Following four lemmas are similar lemmas for reservation station 2.
; See the comment of
;   IU-RS0-src1-INST-tag-LRM-before-if-cntlv-operand0
(defthm exist-uncommitted-LRM-INST-ra-if-IU-RS1-cntlv-operand0
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(IU RS1))
                  (not (b1p (RS-ready1? (IU-RS1 (MA-IU MA)))))
                  (b1p (logbit 0 (cntlv-operand (INST-cntlv i))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (exist-uncommitted-LRM-before-p i (INST-ra i) MT))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

(defthm exist-uncommitted-LRM-INST-ra-if-IU-RS1-cntlv-operand2
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(IU RS1))
                  (not (b1p (RS-ready1? (IU-RS1 (MA-IU MA)))))
```

```
                (b1p (logbit 2 (cntlv-operand (INST-cntlv i))))
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (MAETT-p MT) (MA-state-p MA)
                (INST-p i) (INST-in i MT))
           (exist-uncommitted-LRM-before-p i (INST-rc i) MT))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                :use (:instance consistent-RS-entry-p-if-INST-in))))

(defthm exist-uncommitted-LRM-INST-ra-if-IU-RS1-cntlv-operand3
    (implies (and (inv MT MA)
                (equal (INST-stg i) '(IU RS1))
                (not (b1p (RS-ready1? (IU-RS1 (MA-IU MA)))))
                (b1p (logbit 3 (cntlv-operand (INST-cntlv i))))
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (MAETT-p MT) (MA-state-p MA)
                (INST-p i) (INST-in i MT))
           (exist-uncommitted-LSRM-before-p i (INST-ra i) MT))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                :use (:instance consistent-RS-entry-p-if-INST-in))))

(defthm exist-uncommitted-LRM-INST-rb-if-IU-RS1-cntlv-operand0
    (implies (and (inv MT MA)
                (equal (INST-stg i) '(IU RS1))
                (not (b1p (RS-ready2? (IU-RS1 (MA-IU MA)))))
                (b1p (logbit 0 (cntlv-operand (INST-cntlv i))))
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (MAETT-p MT) (MA-state-p MA)
                (INST-p i) (INST-in i MT))
           (exist-uncommitted-LRM-before-p i (INST-rb i) MT))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                :use (:instance consistent-RS-entry-p-if-INST-in))))

; Src1 field of a reservation station contains the tag to refer to
; the last register modifier of the first operand register.
; See IU-RS0-src1-INST-tag-LRM-before-if-cntlv-operand0.
(defthm IU-RS1-src1-INST-tag-LRM-before-if-cntlv-operand0
    (implies (and (inv MT MA)
                (equal (INST-stg i) '(IU RS1))
                (b1p (logbit 0 (cntlv-operand (INST-cntlv i))))
                (not (b1p (RS-ready1? (IU-RS1 (MA-IU MA)))))
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (MAETT-p MT) (MA-state-p MA)
                (INST-p i) (INST-in i MT))
           (equal (RS-src1 (IU-RS1 (MA-IU MA)))
                  (INST-tag (LRM-before i (INST-ra i) MT))))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                :use (:instance consistent-RS-entry-p-if-INST-in))))

(defthm IU-RS1-src1-INST-tag-LRM-before-if-cntlv-operand2
    (implies (and (inv MT MA)
                (equal (INST-stg i) '(IU RS1))
                (b1p (logbit 2 (cntlv-operand (INST-cntlv i))))
                (not (b1p (RS-ready1? (IU-RS1 (MA-IU MA)))))
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (MAETT-p MT) (MA-state-p MA)
                (INST-p i) (INST-in i MT))
           (equal (RS-src1 (IU-RS1 (MA-IU MA)))
```

```
                  (INST-tag (LRM-before i (INST-rc i) MT))))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

(defthm IU-RS1-src1-INST-tag-LRM-before-if-cntlv-operand3
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(IU RS1))
                  (b1p (logbit 3 (cntlv-operand (INST-cntlv i))))
                  (not (b1p (RS-ready1? (IU-RS1 (MA-IU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (RS-src1 (IU-RS1 (MA-IU MA)))
                  (INST-tag (LSRM-before i (INST-ra i) MT))))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Src2 field of a reservation station contains the tag to refer to
; the last register modifier of the second operand register.
(defthm IU-RS1-src2-INST-tag-LRM-before
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(IU RS1))
                  (b1p (logbit 0 (cntlv-operand (INST-cntlv i))))
                  (not (b1p (RS-ready2? (IU-RS1 (MA-IU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (RS-src2 (IU-RS1 (MA-IU MA)))
                  (INST-tag (LRM-before i (INST-rb i) MT))))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

(encapsulate nil
; local lemma that shows that the value on the CDB is the correct
; source operand value.
(local
(defthm CDB-val-INST-src-val1-if-CDB-ready-for
     (implies (and (equal (INST-stg i) '(IU RS0))
                   (not (b1p (RS-ready1? (IU-RS0 (MA-IU MA)))))
                   (b1p (CDB-ready-for? (RS-src1 (IU-RS0 (MA-IU MA))) MA))
                   (not (b1p (inst-specultv? i)))
                   (not (b1p (INST-modified? i)))
                   (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (INST-p i) (INST-in i MT))
              (equal (equal (CDB-val MA) (INST-src-val1 i)) T))
  :hints (("goal" :in-theory (e/d (CDB-VAL-INST-DEST-VAL
                                   INST-cntlv
                                   INST-IU?
                                   equal-b1p-converter
                                   decode logbit* rdb lift-b-ops
                                   INST-src-val1)
                                  (INST-IU-IF-IU-STG-P))
                  :use (:instance INST-IU-IF-IU-STG-P)))))

(local
(defthm CDB-val-INST-src-val2-if-CDB-ready-for
     (implies (and (equal (INST-stg i) '(IU RS0))
                   (not (b1p (RS-ready2? (IU-RS0 (MA-IU MA)))))
                   (b1p (CDB-ready-for? (RS-src2 (IU-RS0 (MA-IU MA))) MA))
```

```
                  (not (b1p (INST-IU-op? i)))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (equal (CDB-val MA) (INST-src-val2 i)) T))
  :hints (("goal" :in-theory (e/d (CDB-VAL-INST-DEST-VAL
                                   INST-cntlv
                                   INST-IU?
                                   INST-IU-op?
                                   equal-b1p-converter
                                   decode logbit* rdb lift-b-ops
                                   INST-src-val2)
                                  (INST-IU-IF-IU-STG-P))
                  :use (:instance INST-IU-IF-IU-STG-P)))))

; The instruction invariants are preserved when an instruction stays
; in (IU RS0).
(defthm IU-RS0-inst-inv-step-INST-from-IU-RS0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg i) '(IU RS0))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(IU RS0)))
             (IU-RS0-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable lift-b-ops inst-inv-def
                                     step-IU
                                     step-IU-RS0
                                     INST-IU-op?
                                     exception-relations))))
)

(encapsulate nil
(local
(defthm CDB-val-INST-src-val1-if-CDB-ready-for
     (implies (and (equal (INST-stg i) '(IU RS1))
                   (not (b1p (RS-ready1? (IU-RS1 (MA-IU MA)))))
                   (b1p (CDB-ready-for? (RS-src1 (IU-RS1 (MA-IU MA))) MA))
                   (not (b1p (inst-specultv? i)))
                   (not (b1p (INST-modified? i)))
                   (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (INST-p i) (INST-in i MT))
              (equal (equal (CDB-val MA) (INST-src-val1 i)) T))
  :hints (("goal" :in-theory (e/d (CDB-VAL-INST-DEST-VAL
                                   INST-cntlv
                                   INST-IU?
                                   equal-b1p-converter
                                   decode logbit* rdb lift-b-ops
                                   INST-src-val1)
                                  (INST-IU-IF-IU-STG-P))
                  :use (:instance INST-IU-IF-IU-STG-P)))))

(local
(defthm CDB-val-INST-src-val2-if-CDB-ready-for
     (implies (and (equal (INST-stg i) '(IU RS1))
```

```
                  (not (b1p (RS-ready2? (IU-RS1 (MA-IU MA)))))
                  (b1p (CDB-ready-for? (RS-src2 (IU-RS1 (MA-IU MA))) MA))
                  (not (b1p (INST-IU-op? i)))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (equal (CDB-val MA) (INST-src-val2 i)) T))
  :hints (("goal" :in-theory (e/d (CDB-VAL-INST-DEST-VAL
                                   INST-cntlv
                                   INST-IU?
                                   INST-IU-op?
                                   equal-b1p-converter
                                   decode logbit* rdb lift-b-ops
                                   INST-src-val2)
                                  (INST-IU-IF-IU-STG-P))
                  :use (:instance INST-IU-IF-IU-STG-P)))))

; The instruction invariants are preserved when an instruction stays
; in (IU RS1).
(defthm IU-RS1-inst-inv-step-INST-from-IU-RS1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg i) '(IU RS1))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(IU RS1)))
             (IU-RS1-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable lift-b-ops inst-inv-def
                                     step-IU
                                     step-IU-RS1
                                     INST-IU-op?
                                     exception-relations))))
)

; The instruction invariant is preserved for i, if i's next stage is
; (IU RS0).
(defthm IU-RS0-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(IU RS0)))
             (IU-RS0-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :use (:instance stages-reachable-to-IU-RS0))))

; The instruction invariant is preserved for i, if i's next stage is
; (IU RS1).
(defthm IU-RS1-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
```

```
                (INST-p i)
                (INST-in i MT)
                (INST-inv i MA)
                (MT-no-jmp-exintr-before i MT MA sigs)
                (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                (equal (INST-stg (step-INST I MT MA sigs))
                       '(IU RS1)))
           (IU-RS1-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :use (:instance stages-reachable-to-IU-RS1))))

; The instruction invariant is preserved for instruction i if i is
; in the integer unit in the next cycle.
(defthm IU-inst-inv-step-INST
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (MA-input-p sigs)
                (INST-p i)
                (INST-in i MT)
                (INST-inv i MA)
                (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                (MT-no-jmp-exintr-before i MT MA sigs)
                (IU-stg-p (INST-stg (step-INST I MT MA sigs))))
           (IU-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable IU-stg-p IU-inst-inv))))

;;;;;;;;;;Proof of MU-INST-inv;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Proof is similar to CDB-val-INST-src-val1-if-dispatch-IU
(encapsulate nil
(local
(defthm execute-stg-LRM-before
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (INST-p i) (INST-in i MT)
                (exist-uncommitted-LRM-before-p I rname MT)
                (equal (INST-stg i) '(DQ 0))
                (not (b1p (robe-complete?
                           (nth-robe
                            (INST-tag
                             (LRM-before i rname MT))
                            (MA-rob MA))))))
     (execute-stg-p (INST-stg (LRM-before i rname MT))))
  :hints (("goal" :in-theory (disable INST-in-order-LRM-before
                                      INST-is-at-one-of-the-stages
                                      committed-p)
                :use ((:instance INST-is-at-one-of-the-stages
                           (i (LRM-before i rname MT)))
                      (:instance INST-in-order-LRM-before))))))
(local
(defthm execute-stg-LSRM-before
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (INST-p i) (INST-in i MT)
                (exist-uncommitted-LSRM-before-p I rname MT)
                (equal (INST-stg i) '(DQ 0))
                (not (b1p (robe-complete?
                           (nth-robe
                            (INST-tag
                             (LSRM-before i rname MT))
                            (MA-rob MA))))))
     (execute-stg-p (INST-stg (LSRM-before i rname MT))))
  :hints (("goal" :in-theory (disable INST-in-order-LSRM-before
                                      INST-is-at-one-of-the-stages
```

```
                                committed-p)
                 :use ((:instance INST-is-at-one-of-the-stages
                                  (i (LSRM-before i rname MT)))
                       (:instance INST-in-order-LSRM-before))))))

; The value on the CDB is the first operand  for instruction i, when
; CDB-ready-for? is true for DQ-out-tag1.
(defthm CDB-val-INST-src-val1-if-dispatch-MU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-MU? MA))
                  (equal (INST-stg i) '(DQ 0))
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (b1p (CDB-ready-for? (DQ-out-tag1 (MA-DQ MA)) MA))
                  (not (b1p (robe-complete? (nth-robe (DQ-out-tag1 (MA-DQ MA))
                                                      (MA-ROB MA)))))
                  (not (b1p (DQ-out-ready1? (MA-DQ MA)))))
             (equal (CDB-val MA) (INST-src-val1 i)))
  :hints (("goal" :in-theory (enable INST-src-val1
                                     CDB-val-inst-dest-val
                                     DQ-out-tag1
                                     lift-b-ops
                                     INST-cntlv
                                     DISPATCH-inst?
                                     dispatch-to-MU?
                                     DQ-READY-TO-MU?
                                     INST-DECODE-ERROR-DETECTED-P
                                     DQ-out-ready1?))))

; The value on the CDB is the second operand  for instruction i, when
; CDB-ready-for? is true for DQ-out-tag2.
(defthm CDB-val-INST-src-val2-if-dispatch-MU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-MU? MA))
                  (equal (INST-stg i) '(DQ 0))
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (b1p (CDB-ready-for? (DQ-out-tag2 (MA-DQ MA)) MA))
                  (not (b1p (robe-complete? (nth-robe (DQ-out-tag2 (MA-DQ MA))
                                                      (MA-ROB MA)))))
                  (not (b1p (DQ-out-ready2? (MA-DQ MA)))))
             (equal (CDB-val MA) (INST-src-val2 i)))
  :hints (("goal" :in-theory (enable INST-src-val2
                                     CDB-val-inst-dest-val
                                     DQ-out-tag2
                                     lift-b-ops
                                     INST-cntlv
                                     DISPATCH-inst?
                                     dispatch-to-MU?
                                     DQ-READY-TO-MU?
                                     INST-DECODE-ERROR-DETECTED-P
                                     DQ-out-ready2?))))

)
```

```
(in-theory (disable CDB-val-INST-src-val1-if-dispatch-MU
                    CDB-val-INST-src-val2-if-dispatch-MU))

(encapsulate nil
(local
(defthm complete-stg-LRM-before
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (exist-uncommitted-LRM-before-p I rname MT)
                  (equal (INST-stg i) '(DQ 0))
                  (b1p (robe-complete?
                        (nth-robe
                         (INST-tag (LRM-before i rname MT))
                         (MA-rob MA)))))
     (complete-stg-p (INST-stg (LRM-before i rname MT))))
  :hints (("goal" :in-theory (disable INST-in-order-LRM-before
                                      INST-is-at-one-of-the-stages
                                      committed-p)
                  :use ((:instance INST-is-at-one-of-the-stages
                                   (i (LRM-before i rname MT)))
                        (:instance INST-in-order-LRM-before))))))

(local
(defthm complete-stg-LSRM-before
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (exist-uncommitted-LSRM-before-p I rname MT)
                  (equal (INST-stg i) '(DQ 0))
                  (b1p (robe-complete?
                        (nth-robe
                         (INST-tag (LSRM-before i rname MT))
                         (MA-rob MA)))))
     (complete-stg-p (INST-stg (LSRM-before i rname MT))))
  :hints (("goal" :in-theory (disable INST-in-order-LSRM-before
                                      INST-is-at-one-of-the-stages
                                      committed-p)
                  :use ((:instance INST-is-at-one-of-the-stages
                                   (i (LSRM-before i rname MT)))
                        (:instance INST-in-order-LSRM-before)))))
)

; When an instruction is dispatched, its operand may be obtained from
; the ROB, where register values are temporarily stored.  DQ-out-tag1
; designates the ROB entry where the first operand for the dispatched
; instruction is temporarily stored.  The lemma shows that the
; robe-val field of the ROB entry is equal to the first operand value
; for instruction i.
(defthm robe-val-DQ-out-tag1-INST-src-val1-if-dispatch-MU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-MU? MA))
                  (equal (INST-stg i) '(DQ 0))
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (b1p (robe-complete? (nth-robe (DQ-out-tag1 (MA-DQ MA))
                                                 (MA-ROB MA))))
```

```
                (not (b1p (DQ-out-ready1? (MA-DQ MA)))))
            (equal (robe-val (nth-robe (DQ-out-tag1 (MA-DQ MA))
                                       (MA-ROB MA)))
                   (INST-src-val1 i)))
  :hints (("goal" :in-theory (e/d (DQ-out-ready1?
                                   DQ-out-tag1
                                   INST-SRC-val1
                                   INST-cntlv  ; decode rdb logbit*
                                   INST-DECODE-ERROR-DETECTED-P
                                   dispatch-to-MU?
                                   DQ-ready-to-MU?
                                   decode logbit* rdb
                                   exception-relations
                                   INST-exunit-relations
                                   equal-b1p-converter
                                   lift-b-ops)
                                  ()))))

; As discussed in the comment for the previous lemma, operands may be
; obtained from the ROB.  The robe-val field of the ROB entry designated by
; DQ-out-tag2 is equal to the ideal operand value (INST-src-val2 i) for the
; dispatched instruction i.
(defthm robe-val-DQ-out-tag2-INST-src-val2-if-MU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-MU? MA))
                  (equal (INST-stg i) '(DQ 0))
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (b1p (robe-complete? (nth-robe (DQ-out-tag2 (MA-DQ MA))
                                                 (MA-ROB MA))))
                  (not (b1p (DQ-out-ready2? (MA-DQ MA)))))
             (equal (robe-val (nth-robe (DQ-out-tag2 (MA-DQ MA))
                                        (MA-ROB MA)))
                    (INST-src-val2 i)))
  :hints (("goal" :in-theory (e/d (DQ-out-ready2?
                                   DQ-out-tag2
                                   INST-SRC-val2
                                   INST-cntlv
                                   INST-DECODE-ERROR-DETECTED-P
                                   dispatch-to-MU?
                                   DQ-ready-to-MU?
                                   exception-relations
                                   equal-b1p-converter
                                   lift-b-ops)
                                  (ROBE-VAL-DQ-OUT-TAG2-INST-SRC-VAL2-IF-IU)))))
)

; Signal dispatch-val1 from the dispatching logic is the first
; operand value for the dispatched instruction i.
; This combines the results of
;  CDB-val-INST-src-val1-if-dispatch-MU
;  robe-val-DQ-out-tag1-INST-src-val1-if-dispatch-MU
(defthm dispatch-val1-INST-src-val1-if-dispatched-to-MU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (INST-modified? i)))
```

```
                (not (b1p (inst-specultv? i)))
                (not (INST-fetch-error-detected-p i))
                (not (INST-decode-error-detected-p i))
                (b1p (dispatch-to-MU? MA))
                (b1p (dispatch-ready1? MA)))
           (equal (dispatch-val1 MA) (INST-src-val1 i)))
  :hints (("goal" :in-theory (enable dispatch-ready1? dispatch-val1
                                     CDB-val-INST-src-val1-if-dispatch-MU
                                     lift-b-ops))))

; Signal dispatch-val2 from the dispatching logic is the second
; operand value for the dispatched instruction i.
;    CDB-val-INST-src-val2-if-dispatch-MU
;    robe-val-DQ-out-tag2-INST-src-val2-if-MU
(defthm dispatch-val2-INST-src-val2-if-dispatched-to-MU
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (equal (INST-stg i) '(DQ 0))
                (INST-p i) (INST-in i MT)
                (not (b1p (INST-modified? i)))
                (not (b1p (inst-specultv? i)))
                (not (INST-fetch-error-detected-p i))
                (not (INST-decode-error-detected-p i))
                (b1p (dispatch-to-MU? MA))
                (b1p (dispatch-ready2? MA)))
           (equal (dispatch-val2 MA) (INST-src-val2 i)))
  :hints (("goal" :in-theory (enable dispatch-ready2? dispatch-val2
                                     CDB-val-INST-src-val2-if-dispatch-MU
                                     lift-b-ops))))

; The instruction invariants are preserved for instruction that
; moves from DQ0 to MU-RS0.
(defthm MU-RS0-inst-inv-step-INST-from-DQ0
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (MA-input-p sigs)
                (INST-p i) (INST-in i MT)
                (INST-inv i MA)
                (MT-no-jmp-exintr-before i MT MA sigs)
                (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                (equal (INST-stg i) '(DQ 0))
                (equal (INST-stg (step-INST I MT MA sigs))
                       '(MU RS0)))
           (MU-RS0-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (e/d (lift-b-ops inst-inv-def
                                              lift-b-ops
                                              step-MU step-MU-RS0
                                              DQ-ready-to-MU?
                                              INST-MU?
                                              INST-cntlv
                                              equal-b1p-converter
                                              INST-EXCPT-DETECTED-P
                                              exception-relations
                                              dispatch-inst? dispatch-to-MU?
                                              dispatch-cntlv)
                                  (INST-decode-error-if-INST-ra-not-srname-p))
                :cases ((b1p (dispatch-inst? MA))))
          ("subgoal 1" :cases ((b1p (INST-fetch-error? I))))
          ("subgoal 1.2" :cases ((b1p (INST-decode-error? I))))))


; The instruction invariants are preserved for instruction that
```

```
; moves from DQ0 to MU-RS1.
(defthm MU-RS1-inst-inv-step-INST-from-DQ0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i) (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                  (equal (INST-stg i) '(DQ 0))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(MU RS1)))
             (MU-RS1-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
    :hints (("goal" :in-theory (e/d (lift-b-ops inst-inv-def
                                                step-MU step-MU-RS1
                                                INST-cntlv
                                                dispatch-inst?
                                                equal-b1p-converter
                                                INST-EXCPT-DETECTED-P
                                                exception-relations)
                                    ())
                    :cases ((b1p (dispatch-inst? MA))))
            ("subgoal 1" :cases ((b1p (INST-fetch-error? I))))
            ("subgoal 1.2" :cases ((b1p (INST-decode-error? I))))))

; No exception occurs in the MU.
(defthm INST-excpt-detected-p-step-INST-MU
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (MU-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA) (INST-p i)
                  (MA-input-p sigs))
             (not (INST-excpt-detected-p (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (e/d (INST-excpt-detected-p MU-STG-P) ()))))

; Suppose instruction i is stored in a reservation station.
; If the first operand for i is not ready in the reservation station,
; there must be a register modifier of the operand register.
(defthm exist-uncommitted-LRM-INST-ra-if-MU-RS0
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(MU RS0))
                  (not (b1p (RS-ready1? (MU-RS0 (MA-MU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (exist-uncommitted-LRM-before-p i (INST-ra i) MT))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Suppose instruction i is stored in a reservation station.
; If the second operand for i is not ready in the reservation station,
; there must be a register modifier of the operand register.
(defthm exist-uncommitted-LRM-INST-rb-if-MU-RS0
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(MU RS0))
                  (not (b1p (RS-ready2? (MU-RS0 (MA-MU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
```

```
                (INST-p i) (INST-in i MT))
           (exist-uncommitted-LRM-before-p i (INST-rb i) MT))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Similar lemmas for RS1.
(defthm exist-uncommitted-LRM-INST-ra-if-MU-RS1
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(MU RS1))
                  (not (b1p (RS-ready1? (MU-RS1 (MA-MU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (exist-uncommitted-LRM-before-p i (INST-ra i) MT))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Similar lemmas for RS1.
(defthm exist-uncommitted-LRM-INST-rb-if-MU-RS1
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(MU RS1))
                  (not (b1p (RS-ready2? (MU-RS1 (MA-MU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (exist-uncommitted-LRM-before-p i (INST-rb i) MT))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Src1 field of a reservation station RS0 contains the tag referring to
; the last register modifier of the first operand register.
(defthm MU-RS0-src1-INST-tag-LRM-before
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(MU RS0))
                  (not (b1p (RS-ready1? (MU-RS0 (MA-MU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (RS-src1 (MU-RS0 (MA-MU MA)))
                    (INST-tag (LRM-before i (INST-ra i) MT))))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; A similar lemma for RS1.
(defthm MU-RS1-src1-INST-tag-LRM-before
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(MU RS1))
                  (not (b1p (RS-ready1? (MU-RS1 (MA-MU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (RS-src1 (MU-RS1 (MA-MU MA)))
                    (INST-tag (LRM-before i (INST-ra i) MT))))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Src1 field of a reservation station RS0 contains the tag referring to
; the last register modifier of the second operand register.
```

```
(defthm MU-RS0-src2-INST-tag-LRM-before
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(MU RS0))
                  (not (b1p (RS-ready2? (MU-RS0 (MA-MU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (RS-src2 (MU-RS0 (MA-MU MA)))
                    (INST-tag (LRM-before i (INST-rb i) MT))))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; A similar lemma for RS1.
(defthm MU-RS1-src2-INST-tag-LRM-before
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(MU RS1))
                  (not (b1p (RS-ready2? (MU-RS1 (MA-MU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (RS-src2 (MU-RS1 (MA-MU MA)))
                    (INST-tag (LRM-before i (INST-rb i) MT))))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

(encapsulate nil
#|
Following lemmas are for presentation.

The following lemma tells what is (INST-src-val1 i) is for a multiply
instruction.
(defthm CDB-val-INST-src-val1-if-CDB-ready-for-MU-RS0
     (implies (and (equal (INST-stg i) '(MU RS0))
                   (not (b1p (RS-ready1? (MU-RS0 (MA-MU MA)))))
                   (b1p (CDB-ready-for? (RS-src1 (MU-RS0 (MA-MU MA))) MA))
                   (not (b1p (inst-specultv? i)))
                   (not (b1p (INST-modified? i)))
                   (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (INST-p i) (INST-in i MT))
              (equal (read-reg (INST-ra i)
                               (ISA-RF (INST-pre-ISA i)))
                     (INST-src-val1 i)))
  :hints (("goal" :in-theory (e/d (CDB-VAL-INST-DEST-VAL
                                   INST-cntlv
                                   INST-MU?
                                   equal-b1p-converter
                                   decode logbit* rdb lift-b-ops
                                   INST-src-val1)
                                  (INST-MU-IF-MU-STG-P))
                  :use (:instance INST-MU-IF-MU-STG-P))))

|#

(local
(defthm CDB-val-INST-src-val1-if-CDB-ready-for-MU-RS0
     (implies (and (equal (INST-stg i) '(MU RS0))
                   (not (b1p (RS-ready1? (MU-RS0 (MA-MU MA)))))
                   (b1p (CDB-ready-for? (RS-src1 (MU-RS0 (MA-MU MA))) MA))
                   (not (b1p (inst-specultv? i)))
```

```
                  (not (b1p (INST-modified? i)))
                  (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (equal (CDB-val MA) (INST-src-val1 i)) T))
  :hints (("goal" :in-theory (e/d (CDB-VAL-INST-DEST-VAL
                                   INST-cntlv
                                   INST-MU?
                                   equal-b1p-converter
                                   decode logbit* rdb lift-b-ops
                                   INST-src-val1)
                                  (INST-MU-IF-MU-STG-P))
                  :use (:instance INST-MU-IF-MU-STG-P)))))

; This lemma is for presentation.
(local
(defthm CDB-val-INST-src-val1-if-CDB-ready-for-MU-RS0*
     (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (equal (INST-stg i) '(MU RS0))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (b1p (CDB-ready? MA))
                  (equal (CDB-tag MA) (RS-src1 (MU-RS0 (MA-MU MA))))
                  (not (b1p (RS-ready1? (MU-RS0 (MA-MU MA))))))
             (equal (CDB-val MA)
                    (INST-src-val1 i)))
  :hints (("goal" :in-theory (e/d (equal-b1p-converter
                                   CDB-ready-for?) ())))
  :rule-classes nil))

(local
(defthm CDB-val-INST-src-val2-if-CDB-ready-for-MU-RS0
     (implies (and (equal (INST-stg i) '(MU RS0))
                  (not (b1p (RS-ready2? (MU-RS0 (MA-MU MA)))))
                  (b1p (CDB-ready-for? (RS-src2 (MU-RS0 (MA-MU MA))) MA))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (equal (CDB-val MA) (INST-src-val2 i)) T))
  :hints (("goal" :in-theory (e/d (CDB-VAL-INST-DEST-VAL
                                   INST-cntlv
                                   INST-IU?
                                   INST-IU-op?
                                   equal-b1p-converter
                                   decode logbit* rdb lift-b-ops
                                   INST-src-val2)
                                  (INST-IU-IF-IU-STG-P))
                  :use (:instance INST-IU-IF-IU-STG-P)))))

; The instruction invariants are preserved for instruction i that stays
; in MU-RS0.
(defthm MU-RS0-inst-inv-step-INST-from-MU-RS0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i) (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
```

```
                (equal (INST-stg i) '(MU RS0))
                (equal (INST-stg (step-INST I MT MA sigs))
                       '(MU RS0)))
           (MU-RS0-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable lift-b-ops inst-inv-def
                                     step-MU
                                     step-MU-RS0
                                     exception-relations))))
)

(encapsulate nil
(local
(defthm CDB-val-INST-src-val1-if-CDB-ready-for-MU-RS1
     (implies (and (equal (INST-stg i) '(MU RS1))
                   (not (b1p (RS-ready1? (MU-RS1 (MA-MU MA)))))
                   (b1p (CDB-ready-for? (RS-src1 (MU-RS1 (MA-MU MA))) MA))
                   (not (b1p (inst-specultv? i)))
                   (not (b1p (INST-modified? i)))
                   (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (INST-p i) (INST-in i MT))
              (equal (equal (CDB-val MA) (INST-src-val1 i)) T))
  :hints (("goal" :in-theory (e/d (CDB-VAL-INST-DEST-VAL
                                   INST-cntlv
                                   INST-MU?
                                   equal-b1p-converter
                                   decode logbit* rdb lift-b-ops
                                   INST-src-val1)
                                  (INST-MU-IF-MU-STG-P))
                  :use (:instance INST-MU-IF-MU-STG-P)))))

(local
(defthm CDB-val-INST-src-val2-if-CDB-ready-for-MU-RS1
     (implies (and (equal (INST-stg i) '(MU RS1))
                   (not (b1p (RS-ready2? (MU-RS1 (MA-MU MA)))))
                   (b1p (CDB-ready-for? (RS-src2 (MU-RS1 (MA-MU MA))) MA))
                   (not (b1p (inst-specultv? i)))
                   (not (b1p (INST-modified? i)))
                   (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (INST-p i) (INST-in i MT))
              (equal (equal (CDB-val MA) (INST-src-val2 i)) T))
  :hints (("goal" :in-theory (e/d (CDB-VAL-INST-DEST-VAL
                                   INST-cntlv
                                   INST-IU?
                                   INST-IU-op?
                                   equal-b1p-converter
                                   decode logbit* rdb lift-b-ops
                                   INST-src-val2)
                                  (INST-IU-IF-IU-STG-P))
                  :use (:instance INST-IU-IF-IU-STG-P)))))

; The instruction invariants are preserved for instruction i that stays
; in MU-RS1.
(defthm MU-RS1-inst-inv-step-INST-from-MU-RS1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i) (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg i) '(MU RS1))
```

```
                (equal (INST-stg (step-INST I MT MA sigs))
                       '(MU RS1)))
           (MU-RS1-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable lift-b-ops inst-inv-def
                                     step-MU
                                     step-MU-RS1
                                     exception-relations))))
)

; The instruction invariants are preserved for instruction i whose
; next state is MU-RS0.
(defthm MU-RS0-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i) (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(MU RS0)))
             (MU-RS0-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :use (:instance stages-reachable-to-MU-RS0))))

; The instruction invariants are preserved for instruction i whose
; next state is MU-RS1.
(defthm MU-RS1-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i) (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(MU RS1)))
             (MU-RS1-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
    :hints (("goal" :use (:instance stages-reachable-to-MU-RS1))))

; Instruction invariants are preserved for instruction i that moves
; from MU-RS0 to MU-lch1.
(defthm MU-lch1-inst-inv-step-INST-from-MU-RS0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i) (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg i) '(MU RS0))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(MU lch1)))
             (MU-lch1-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable lift-b-ops inst-inv-def
                                     step-MU
                                     step-MU-lch1
                                     issue-MU-RS0? MU-RS1-ISSUE-READY?
                                     issue-MU-RS1? MU-RS0-ISSUE-READY?
                                     exception-relations))))

; Instruction invariants are preserved for instruction i that moves
; from MU-RS1 to MU-lch1.
(defthm MU-lch1-inst-inv-step-INST-from-MU-RS1
```

```
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i) (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg i) '(MU RS1))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(MU lch1)))
             (MU-lch1-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable lift-b-ops inst-inv-def
                                     step-MU
                                     step-MU-lch1
                                     issue-MU-RS0? MU-RS1-ISSUE-READY?
                                     issue-MU-RS1? MU-RS0-ISSUE-READY?
                                     exception-relations))))

; Instruction invariants are preserved for instruction i that stays
; in MU-lch1.
(defthm MU-lch1-inst-inv-step-INST-from-MU-lch1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i) (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg i) '(MU lch1))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(MU lch1)))
             (MU-lch1-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable lift-b-ops inst-inv-def
                                     step-MU
                                     step-MU-lch1
                                     exception-relations))))

; Instruction invariants are preserved for instruction i whose next state
; is MU-lch1.
(defthm MU-lch1-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i) (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(MU lch1)))
             (MU-lch1-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :use (:instance stages-reachable-to-MU-lch1))))

; Instruction invariants are preserved for instruction i that moves from
; MU-lch1 to MU-lch2.
(defthm MU-lch2-inst-inv-step-INST-from-MU-lch1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i) (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg i) '(MU lch1))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(MU lch2)))
             (MU-lch2-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
```

```
  :hints (("goal" :in-theory (enable lift-b-ops inst-inv-def
                                     step-MU
                                     step-MU-lch2
                                     exception-relations))))

; Instruction invariants are preserved for instruction i that stays
; in MU-lch2.
(defthm MU-lch2-inst-inv-step-INST-from-MU-lch2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i) (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg i) '(MU lch2))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(MU lch2)))
             (MU-lch2-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable lift-b-ops inst-inv-def
                                     step-MU
                                     step-MU-lch2
                                     exception-relations))))

; Instruction invariants are preserved for instruction i whose next stage
; is MU-lch2.
(defthm MU-lch2-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i) (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(MU lch2)))
             (MU-lch2-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :use (:instance stages-reachable-to-MU-lch2))))

; A landmark lemma.  The instruction invariants are preserved for
; instruction i whose next state is in the MU.
(defthm MU-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i) (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                  (MU-stg-p (INST-stg (step-INST I MT MA sigs))))
             (MU-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable MU-stg-p MU-inst-inv))))

;;;;;;;;;;;;;;;;;;;;Proof of BU-inst-inv;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(encapsulate nil
(local
(defthm complete-stg-LRM-before
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (exist-uncommitted-LRM-before-p I rname MT)
                  (equal (INST-stg i) '(DQ 0))
                  (b1p (robe-complete?
                        (nth-robe
```

```
                        (INST-tag (LRM-before i rname MT))
                        (MA-rob MA)))))
     (complete-stg-p (INST-stg (LRM-before i rname MT))))
  :hints (("goal" :in-theory (disable INST-in-order-LRM-before
                                      INST-is-at-one-of-the-stages
                                      committed-p)
                  :use ((:instance INST-is-at-one-of-the-stages
                                   (i (LRM-before i rname MT)))
                        (:instance INST-in-order-LRM-before))))))

; As discussed in the comment for the previous lemma, operands may be
; obtained from the ROB.  The robe-val field of the ROB entry designated by
; DQ-out-tag3 contains the correct source operand value for the
; dispatched instruction i.
(defthm robe-val-DQ-out-tag3-INST-src-val3-if-BU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-BU? MA))
                  (equal (INST-stg i) '(DQ 0))
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (b1p (robe-complete? (nth-robe (DQ-out-tag3 (MA-DQ MA))
                                                 (MA-ROB MA))))
                  (not (b1p (DQ-out-ready3? (MA-DQ MA)))))
             (equal (robe-val (nth-robe (DQ-out-tag3 (MA-DQ MA))
                                        (MA-ROB MA)))
                    (INST-src-val3 i)))
  :hints (("goal" :in-theory (e/d (DQ-out-ready3?
                                   DQ-out-tag3
                                   INST-SRC-val3
                                   INST-cntlv
                                   INST-DECODE-ERROR-DETECTED-P
                                   dispatch-to-BU?
                                   DQ-ready-to-BU?
                                   decode logbit* rdb
                                   exception-relations
                                   equal-b1p-converter
                                   lift-b-ops)
                                  ()))))
)

; Proof is similar to CDB-val-INST-src-val1-if-dispatch-IU
(encapsulate nil
(local
(defthm execute-stg-LRM-before
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (exist-uncommitted-LRM-before-p I rname MT)
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (robe-complete?
                             (nth-robe
                              (INST-tag
                               (LRM-before i rname MT))
                              (MA-rob MA))))))
     (execute-stg-p (INST-stg (LRM-before i rname MT))))
  :hints (("goal" :in-theory (disable INST-in-order-LRM-before
                                      INST-is-at-one-of-the-stages
                                      committed-p)
```

```
                  :use ((:instance INST-is-at-one-of-the-stages
                                   (i (LRM-before i rname MT)))
                        (:instance INST-in-order-LRM-before))))))

; The value on the CDB is the third operand value for instruction i, when
; CDB-ready-for? is true for DQ-out-tag3.
(defthm CDB-val-INST-src-val-if-dispatch-BU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-BU? MA))
                  (equal (INST-stg i) '(DQ 0))
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (b1p (CDB-ready-for? (DQ-out-tag3 (MA-DQ MA)) MA))
                  (not (b1p (robe-complete? (nth-robe (DQ-out-tag3 (MA-DQ MA))
                                                      (MA-ROB MA)))))
                  (not (b1p (DQ-out-ready3? (MA-DQ MA)))))
             (equal (CDB-val MA) (INST-src-val3 i)))
  :hints (("goal" :in-theory (enable INST-src-val1
                                     cdb-val-inst-dest-val
                                     INST-src-val3
                                     DQ-out-tag3
                                     lift-b-ops
                                     INST-cntlv
                                     DISPATCH-inst?
                                     dispatch-to-BU?
                                     DQ-READY-TO-BU?
                                     INST-DECODE-ERROR-DETECTED-P
                                     DQ-out-ready3?))))
)
(in-theory (disable CDB-val-INST-src-val-if-dispatch-BU))

; An output from the dispatching logic dispatch-val3 is the third
; operand of the dispatched instruction i.
; This combines the results of two lemmas:
;  robe-val-DQ-out-tag3-INST-src-val3-if-BU
;  CDB-val-INST-src-val-if-dispatch-BU
(defthm dispatch-val3-INST-src-val1-if-dispatched-to-BU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (b1p (dispatch-to-BU? MA))
                  (b1p (dispatch-ready3? MA)))
             (equal (dispatch-val3 MA) (INST-src-val3 i)))
  :hints (("goal" :in-theory (enable dispatch-ready3? dispatch-val3
                                     CDB-val-INST-src-val-if-dispatch-BU
                                     lift-b-ops))))

; The instruction invariants are preserved for instruction i that moves
; from DQ0 to BU-RS0.
(defthm BU-RS0-inst-inv-step-INST-from-DQ0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
```

```
                (INST-p i)
                (INST-in i MT)
                (INST-inv i MA)
                (MT-no-jmp-exintr-before i MT MA sigs)
                (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                (equal (INST-stg I) '(DQ 0))
                (equal (INST-stg (step-INST I MT MA sigs))
                       '(BU RS0)))
           (BU-RS0-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (e/d (lift-b-ops inst-inv-def
                                              step-BU step-BU-RS0
                                              INST-cntlv
                                              dispatch-inst?
                                              equal-b1p-converter
                                              INST-EXCPT-DETECTED-P
                                              exception-relations)
                                  ())
                  :cases ((b1p (dispatch-inst? MA))))
          ("subgoal 1" :cases ((b1p (INST-fetch-error? I))))
          ("subgoal 1.2" :cases ((b1p (INST-decode-error? I))))))

; The instruction invariants are preserved for instruction i that moves
; from DQ0 to BU-RS1.
(defthm BU-RS1-inst-inv-step-INST-from-DQ0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                  (equal (INST-stg I) '(DQ 0))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(BU RS1)))
           (BU-RS1-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (e/d (lift-b-ops inst-inv-def
                                              step-BU step-BU-RS1
                                              INST-cntlv
                                              dispatch-inst?
                                              equal-b1p-converter
                                              INST-EXCPT-DETECTED-P
                                              exception-relations)
                                  ())
                  :cases ((b1p (dispatch-inst? MA))))
          ("subgoal 1" :cases ((b1p (INST-fetch-error? I))))
          ("subgoal 1.2" :cases ((b1p (INST-decode-error? I))))))

; Exception does not occur in the BU.
(defthm INST-excpt-detected-p-step-INST-BU
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (BU-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA) (INST-p i)
                  (MA-input-p sigs))
           (not (INST-excpt-detected-p (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (e/d (INST-excpt-detected-p BU-STG-P) ()))))

; Suppose instruction i is stored in a reservation station.
; If the first operand for i is not ready in the reservation station,
```

```
; there must be a register modifier of the operand register.
(defthm exist-uncommitted-LRM-INST-rc-if-BU-RS0
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(BU RS0))
                  (not (b1p (BU-RS-ready? (BU-RS0 (MA-BU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (exist-uncommitted-LRM-before-p i (INST-rc i) MT))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Similar to exist-uncommitted-LRM-INST-rc-if-BU-RS0.
(defthm exist-uncommitted-LRM-INST-rc-if-BU-RS1
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(BU RS1))
                  (not (b1p (BU-RS-ready? (BU-RS1 (MA-BU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (exist-uncommitted-LRM-before-p i (INST-rc i) MT))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Src field of a reservation station contains the tag to refer to
; the last register modifier of the operand register.
(defthm BU-RS0-src-INST-tag-LRM-before
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(BU RS0))
                  (not (b1p (BU-RS-ready? (BU-RS0 (MA-BU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (BU-RS-src (BU-RS0 (MA-BU MA)))
                    (INST-tag (LRM-before i (INST-rc i) MT))))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Similar to BU-RS0-src-INST-tag-LRM-before.
(defthm BU-RS1-src1-INST-tag-LRM-before
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(BU RS1))
                  (not (b1p (BU-RS-ready? (BU-RS1 (MA-BU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (BU-RS-src (BU-RS1 (MA-BU MA)))
                    (INST-tag (LRM-before i (INST-rc i) MT))))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

(encapsulate nil
(local
(defthm CDB-val-INST-src-val3-if-CDB-ready-for-BU-RS0
     (implies (and (equal (INST-stg i) '(BU RS0))
                   (not (b1p (BU-RS-ready? (BU-RS0 (MA-BU MA)))))
                   (b1p (CDB-ready-for? (BU-RS-src (BU-RS0 (MA-BU MA))) MA))
                   (not (b1p (inst-specultv? i)))
```

```
                  (not (b1p (INST-modified? i)))
                  (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (equal (CDB-val MA) (INST-src-val3 i)) T))
  :hints (("goal" :in-theory (e/d (CDB-VAL-INST-DEST-VAL
                                   INST-cntlv
                                   INST-BU?
                                   equal-b1p-converter
                                   decode logbit* rdb lift-b-ops
                                   INST-src-val3)
                                  (INST-BU-IF-BU-STG-P))
                  :use (:instance INST-BU-IF-BU-STG-P)))))

; The instruction invariants are preserved for instruction i that stays
; in BU-RS0.
(defthm BU-RS0-inst-inv-step-INST-from-BU-RS0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg I) '(BU RS0))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(BU RS0)))
             (BU-RS0-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable lift-b-ops inst-inv-def
                                     step-BU
                                     step-BU-RS0
                                     exception-relations))))
)

(encapsulate nil
(local
(defthm CDB-val-INST-src-val3-if-CDB-ready-for-BU-RS1
     (implies (and (equal (INST-stg i) '(BU RS1))
                  (not (b1p (BU-RS-ready? (BU-RS1 (MA-BU MA)))))
                  (b1p (CDB-ready-for? (BU-RS-src (BU-RS1 (MA-BU MA))) MA))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (equal (CDB-val MA) (INST-src-val3 i)) T))
  :hints (("goal" :in-theory (e/d (CDB-VAL-INST-DEST-VAL
                                   INST-cntlv
                                   INST-BU?
                                   equal-b1p-converter
                                   decode logbit* rdb lift-b-ops
                                   INST-src-val3)
                                  (INST-BU-IF-BU-STG-P))
                  :use (:instance INST-BU-IF-BU-STG-P)))))

; The instruction invariants are preserved for instruction i that stays
; in BU-RS1.
(defthm BU-RS1-inst-inv-step-INST-from-BU-RS1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
```

```
                (INST-in i MT)
                (INST-inv i MA)
                (MT-no-jmp-exintr-before i MT MA sigs)
                (equal (INST-stg I) '(BU RS1))
                (equal (INST-stg (step-INST I MT MA sigs))
                       '(BU RS1)))
           (BU-RS1-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable lift-b-ops inst-inv-def
                                     step-BU
                                     step-BU-RS1
                                     exception-relations))))
)

; The instruction invariants are preserved for instruction i whose next
; stage is BU-RS0.
(defthm BU-RS0-inst-inv-step-INST
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (MA-input-p sigs)
                (INST-p i)
                (INST-in i MT)
                (INST-inv i MA)
                (MT-no-jmp-exintr-before i MT MA sigs)
                (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                (equal (INST-stg (step-INST I MT MA sigs))
                       '(BU RS0)))
           (BU-RS0-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :use (:instance stages-reachable-to-BU-RS0))))

; The instruction invariants are preserved for instruction i whose next
; stage is BU-RS1.
(defthm BU-RS1-inst-inv-step-INST
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (MA-input-p sigs)
                (INST-p i)
                (INST-in i MT)
                (INST-inv i MA)
                (MT-no-jmp-exintr-before i MT MA sigs)
                (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                (equal (INST-stg (step-INST I MT MA sigs))
                       '(BU RS1)))
           (BU-RS1-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :use (:instance stages-reachable-to-BU-RS1))))

; The instruction invariants are preserved for instruction i which will be
; in BU in the next cycle.
(defthm BU-inst-inv-step-INST
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (MA-input-p sigs)
                (INST-p i)
                (INST-in i MT)
                (INST-inv i MA)
                (MT-no-jmp-exintr-before i MT MA sigs)
                (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                (BU-stg-p (INST-stg (step-INST I MT MA sigs))))
           (BU-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable BU-stg-p BU-inst-inv))))

;;;;;;;;;;;;;;;;;;Proof of LSU-inst-inv-step-INST;;;;;;;;;;;;;;;;
; The proof of LSU-inst-inv-step-INST has following subgoals.
```

```
;   LSU-RS0-inst-inv-step-INST
;   LSU-RS1-inst-inv-step-INST
;   LSU-wbuf1-inst-inv-step-INST
;   LSU-wbuf0-inst-inv-step-INST
;   LSU-rbuf-inst-inv-step-INST
;   LSU-wbuf0-lch-inst-inv-step-INST
;   LSU-wbuf1-lch-inst-inv-step-INST
;   LSU-lch-inst-inv-step-INST
; LSU-lch-inst-inv-step-INST takes a considerable amount of work and
; this includes the proof for the load-bypassing and load-hoisting.

(encapsulate nil
(local
(defthm complete-stg-LRM-before
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (exist-uncommitted-LRM-before-p I rname MT)
                  (equal (INST-stg i) '(DQ 0))
                  (b1p (robe-complete?
                        (nth-robe
                         (INST-tag (LRM-before i rname MT))
                         (MA-rob MA)))))
     (complete-stg-p (INST-stg (LRM-before i rname MT))))
  :hints (("goal" :in-theory (disable INST-in-order-LRM-before
                                      INST-is-at-one-of-the-stages
                                      committed-p)
                  :use ((:instance INST-is-at-one-of-the-stages
                                   (i (LRM-before i rname MT)))
                        (:instance INST-in-order-LRM-before))))))

; The robe-val field of the ROB entry designated by DQ-out-tag1
; contains the correct source operand value for the dispatched
; instruction i.
(defthm robe-val-DQ-out-tag1-INST-src-val1-if-LSU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-LSU? MA))
                  (equal (INST-stg i) '(DQ 0))
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (b1p (robe-complete? (nth-robe (DQ-out-tag1 (MA-DQ MA))
                                                 (MA-ROB MA))))
                  (not (b1p (DQ-out-ready1? (MA-DQ MA)))))
             (equal (robe-val (nth-robe (DQ-out-tag1 (MA-DQ MA))
                                        (MA-ROB MA)))
                    (INST-src-val1 i)))
  :hints (("goal" :in-theory (e/d (DQ-out-ready1?
                                   DQ-out-tag1
                                   INST-SRC-val1
                                   INST-cntlv
                                   INST-DECODE-ERROR-DETECTED-P
                                   dispatch-to-LSU?
                                   DQ-ready-to-LSU?
                                   decode logbit* rdb
                                   exception-relations
                                   equal-b1p-converter
                                   lift-b-ops)
                                  (RETIRED-DISPATCHED-INST
```

```
                                  UNIQ-INST-OF-TAG-IF-ROBE-VALID
                                  ROBE-COMPLETE-NTH-ROBE-RIX
                                  COMPLETED-DISPATCHED-INST)))))

; The robe-val field of the ROB entry designated by DQ-out-tag2
; contains the correct source operand value for the dispatched
; instruction i.
(defthm robe-val-DQ-out-tag2-INST-src-val2-if-LSU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-LSU? MA))
                  (equal (INST-stg i) '(DQ 0))
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (b1p (robe-complete? (nth-robe (DQ-out-tag2 (MA-DQ MA))
                                                 (MA-ROB MA))))
                  (not (b1p (DQ-out-ready2? (MA-DQ MA)))))
             (equal (robe-val (nth-robe (DQ-out-tag2 (MA-DQ MA))
                                        (MA-ROB MA)))
                    (INST-src-val2 i)))
  :hints (("goal" :in-theory (e/d (DQ-out-ready2?
                                   DQ-out-tag2
                                   INST-SRC-val2
                                   INST-cntlv
                                   INST-DECODE-ERROR-DETECTED-P
                                   dispatch-to-LSU?
                                   DQ-ready-to-LSU?
                                   decode logbit* rdb
                                   exception-relations
                                   equal-b1p-converter
                                   lift-b-ops)
                                  (RETIRED-DISPATCHED-INST
                                   UNIQ-INST-OF-TAG-IF-ROBE-VALID
                                   ROBE-COMPLETE-NTH-ROBE-RIX
                                   COMPLETED-DISPATCHED-INST)))))

; The robe-val field of the ROB entry designated by DQ-out-tag3
; contains the correct source operand value for the dispatched
; instruction i.
(defthm robe-val-DQ-out-tag3-INST-src-val3-if-LSU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-LSU? MA))
                  (equal (INST-stg i) '(DQ 0))
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (b1p (robe-complete? (nth-robe (DQ-out-tag3 (MA-DQ MA))
                                                 (MA-ROB MA))))
                  (not (b1p (DQ-out-ready3? (MA-DQ MA)))))
             (equal (robe-val (nth-robe (DQ-out-tag3 (MA-DQ MA))
                                        (MA-ROB MA)))
                    (INST-src-val3 i)))
  :hints (("goal" :in-theory (e/d (DQ-out-ready3?
                                   DQ-out-tag3
                                   INST-SRC-val3
                                   INST-cntlv
```

```
                                  INST-DECODE-ERROR-DETECTED-P
                                  decode logbit* rdb
                                  exception-relations
                                  equal-b1p-converter
                                  lift-b-ops)
                                 (RETIRED-DISPATCHED-INST
                                  UNIQ-INST-OF-TAG-IF-ROBE-VALID
                                  ROBE-COMPLETE-NTH-ROBE-RIX
                                  COMPLETED-DISPATCHED-INST)))))
)

(encapsulate nil
(local
(defthm execute-stg-LRM-before
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (exist-uncommitted-LRM-before-p I rname MT)
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (robe-complete?
                             (nth-robe
                              (INST-tag
                               (LRM-before i rname MT))
                              (MA-rob MA))))))
     (execute-stg-p (INST-stg (LRM-before i rname MT))))
  :hints (("goal" :in-theory (disable INST-in-order-LRM-before
                                      INST-is-at-one-of-the-stages
                                      committed-p)
                  :use ((:instance INST-is-at-one-of-the-stages
                                   (i (LRM-before i rname MT)))
                        (:instance INST-in-order-LRM-before))))))

; The value obtained from CDB is the correct operand value for instruction i.
(defthm CDB-val-INST-src-val1-if-dispatch-LSU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-LSU? MA))
                  (equal (INST-stg i) '(DQ 0))
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (b1p (CDB-ready-for? (DQ-out-tag1 (MA-DQ MA)) MA))
                  (not (b1p (robe-complete? (nth-robe (DQ-out-tag1 (MA-DQ MA))
                                                      (MA-ROB MA)))))
                  (not (b1p (DQ-out-ready1? (MA-DQ MA)))))
             (equal (CDB-val MA) (INST-src-val1 i)))
  :hints (("goal" :in-theory (e/d (INST-src-val1
                                   cdb-val-inst-dest-val
                                   INST-src-val1
                                   DQ-out-tag1
                                   lift-b-ops
                                   INST-cntlv
                                   DISPATCH-inst?
                                   dispatch-to-LSU?
                                   DQ-READY-TO-LSU?
                                   INST-DECODE-ERROR-DETECTED-P
                                   DQ-out-ready1?)
                                  (RETIRED-DISPATCHED-INST
                                   UNIQ-INST-OF-TAG-IF-ROBE-VALID
                                   ROBE-COMPLETE-NTH-ROBE-RIX
```

```
                                  COMPLETED-DISPATCHED-INST)))))

; The value obtained from CDB is the correct operand value for instruction i.
(defthm CDB-val-INST-src-val2-if-dispatch-LSU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-LSU? MA))
                  (equal (INST-stg i) '(DQ 0))
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (b1p (CDB-ready-for? (DQ-out-tag2 (MA-DQ MA)) MA))
                  (not (b1p (robe-complete? (nth-robe (DQ-out-tag2 (MA-DQ MA))
                                                      (MA-ROB MA)))))
                  (not (b1p (DQ-out-ready2? (MA-DQ MA)))))
             (equal (CDB-val MA) (INST-src-val2 i)))
  :hints (("goal" :in-theory (e/d (INST-src-val2
                                   cdb-val-inst-dest-val
                                   INST-src-val2
                                   DQ-out-tag2
                                   lift-b-ops
                                   INST-cntlv
                                   DISPATCH-inst?
                                   dispatch-to-LSU?
                                   DQ-READY-TO-LSU?
                                   INST-DECODE-ERROR-DETECTED-P
                                   DQ-out-ready2?)
                                  (RETIRED-DISPATCHED-INST
                                   UNIQ-INST-OF-TAG-IF-ROBE-VALID
                                   ROBE-COMPLETE-NTH-ROBE-RIX
                                   COMPLETED-DISPATCHED-INST)))))

; The value obtained from CDB is the correct operand value for instruction i.
(defthm CDB-val-INST-src-val3-if-dispatch-LSU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-to-LSU? MA))
                  (equal (INST-stg i) '(DQ 0))
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (b1p (CDB-ready-for? (DQ-out-tag3 (MA-DQ MA)) MA))
                  (not (b1p (robe-complete? (nth-robe (DQ-out-tag3 (MA-DQ MA))
                                                      (MA-ROB MA)))))
                  (not (b1p (DQ-out-ready3? (MA-DQ MA)))))
             (equal (CDB-val MA) (INST-src-val3 i)))
  :hints (("goal" :in-theory (e/d (INST-src-val3
                                   cdb-val-inst-dest-val
                                   INST-src-val3
                                   DQ-out-tag3
                                   lift-b-ops
                                   INST-cntlv
                                   DISPATCH-inst?
                                   dispatch-to-LSU?
                                   DQ-READY-TO-LSU?
                                   INST-DECODE-ERROR-DETECTED-P
                                   DQ-out-ready3?)
                                  (RETIRED-DISPATCHED-INST
```

```
                                  UNIQ-INST-OF-TAG-IF-ROBE-VALID
                                  ROBE-COMPLETE-NTH-ROBE-RIX
                                  COMPLETED-DISPATCHED-INST)))))
)
(in-theory (disable CDB-val-INST-src-val1-if-dispatch-LSU
                    CDB-val-INST-src-val2-if-dispatch-LSU
                    CDB-val-INST-src-val3-if-dispatch-LSU))

; An output signal dispatch-val1 from the dispatching logic is the first
; operand of the dispatched instruction i.
; This combines the result of
;   robe-val-DQ-out-tag1-INST-src-val1-if-LSU
;   CDB-val-INST-src-val1-if-dispatch-LSU
(defthm dispatch-val1-INST-src-val1-if-dispatched-to-LSU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (b1p (dispatch-to-LSU? MA))
                  (b1p (dispatch-ready1? MA)))
             (equal (dispatch-val1 MA) (INST-src-val1 i)))
  :hints (("goal" :in-theory (enable dispatch-ready1? dispatch-val1
                                     CDB-val-INST-src-val1-if-dispatch-LSU
                                     lift-b-ops))))

; An output signal dispatch-val2 from the dispatching logic is the second
; operand of the dispatched instruction i.
; This combines the result of
;    robe-val-DQ-out-tag2-INST-src-val2-if-LSU
;    CDB-val-INST-src-val2-if-dispatch-LSU
(defthm dispatch-val2-INST-src-val2-if-dispatched-to-LSU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (b1p (dispatch-to-LSU? MA))
                  (b1p (dispatch-ready2? MA)))
             (equal (dispatch-val2 MA) (INST-src-val2 i)))
  :hints (("goal" :in-theory (enable dispatch-ready2? dispatch-val2
                                     CDB-val-INST-src-val2-if-dispatch-LSU
                                     lift-b-ops))))

; An output signal dispatch-val3 from the dispatching logic is the third
; operand of the dispatched instruction i.
; This combines the result of
;   robe-val-DQ-out-tag3-INST-src-val3-if-LSU
;   CDB-val-INST-src-val3-if-dispatch-LSU
(defthm dispatch-val3-INST-src-val3-if-dispatched-to-LSU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
```

```
                (not (INST-fetch-error-detected-p i))
                (not (INST-decode-error-detected-p i))
                (b1p (dispatch-to-LSU? MA))
                (b1p (dispatch-ready3? MA)))
           (equal (dispatch-val3 MA) (INST-src-val3 i)))
  :hints (("goal" :in-theory (enable dispatch-ready3? dispatch-val3
                                     CDB-val-INST-src-val3-if-dispatch-LSU
                                     lift-b-ops))))

; The operand type of the load store instruction is immediate (designated
; by (INST-LSU-op? i)), dispatch-ready1 is 1.
(defthm dispatch-ready1-if-INST-LSU-op
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(DQ 0))
                (b1p (INST-LSU-op? i))
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (INST-fetch-error-detected-p I)))
           (equal (dispatch-ready1? MA) 1))
  :hints (("goal" :in-theory (enable dispatch-ready1? lift-b-ops
                                     DQ-out-ready1? INST-LSU-OP?
                                     INST-cntlv
                                     equal-b1p-converter))))

; The instruction invariants are preserved if i moves from DQ0 to LSU-RS0.
(defthm LSU-RS0-inst-inv-step-INST-from-DQ0
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (MA-input-p sigs)
                (INST-p i)
                (INST-in i MT)
                (INST-inv i MA)
                (MT-no-jmp-exintr-before i MT MA sigs)
                (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                (equal (INST-stg I) '(DQ 0))
                (equal (INST-stg (step-INST I MT MA sigs))
                       '(LSU RS0)))
           (LSU-RS0-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (e/d (lift-b-ops inst-inv-def
                                              step-LSU step-LSU-RS0
                                              INST-cntlv
                                              dispatch-inst?
                                              equal-b1p-converter
                                              INST-EXCPT-DETECTED-P
                                              exception-relations
                                              INST-LSU-op? INST-ld-st?)
                                  ())
                :cases ((b1p (dispatch-inst? MA))))
          ("subgoal 1" :cases ((b1p (INST-fetch-error? I))))
          ("subgoal 1.2" :cases ((b1p (INST-decode-error? I))))))

; The instruction invariants are preserved if i moves from DQ0 to LSU-RS1.
(defthm LSU-RS1-inst-inv-step-INST-from-DQ0
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (MA-input-p sigs)
                (INST-p i)
                (INST-in i MT)
                (INST-inv i MA)
                (MT-no-jmp-exintr-before i MT MA sigs)
```

```
                (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                (equal (INST-stg I) '(DQ 0))
                (equal (INST-stg (step-INST I MT MA sigs))
                       '(LSU RS1)))
           (LSU-RS1-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (e/d (lift-b-ops inst-inv-def
                                              step-LSU step-LSU-RS1
                                              INST-cntlv
                                              dispatch-inst?
                                              equal-b1p-converter
                                              INST-EXCPT-DETECTED-P
                                              exception-relations
                                              INST-LSU-op? INST-ld-st?)
                                  ())
                :cases ((b1p (dispatch-inst? MA))))
          ("subgoal 1" :cases ((b1p (INST-fetch-error? I))))
          ("subgoal 1.2" :cases ((b1p (INST-decode-error? I))))))

; No exception is detected at the reservation stations of LSU.
(defthm INST-excpt-detected-p-step-INST-LSU
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (or (equal (INST-stg i) '(LSU RS0))
                      (equal (INST-stg i) '(LSU RS1)))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA) (INST-p i)
                  (MA-input-p sigs))
             (not (INST-excpt-detected-p (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (e/d (INST-excpt-detected-p LSU-STG-P
                                   INST-data-accs-error-detected-p
                                   INST-load-accs-error-detected-p
                                   INST-store-accs-error-detected-p) ()))))

; Suppose instruction i is stored in a reservation station.
; If the first operand for i is not ready in the reservation station,
; there must be a register modifier of the operand register.
(defthm exist-uncommitted-LRM-INST-ra-if-LSU-RS0
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(LSU RS0))
                  (not (b1p (LSU-RS-rdy1? (LSU-RS0 (MA-LSU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (exist-uncommitted-LRM-before-p i (INST-ra i) MT))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Similar to exist-uncommitted-LRM-INST-ra-if-LSU-RS0.
(defthm exist-uncommitted-LRM-INST-ra-if-LSU-RS1
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(LSU RS1))
                  (not (b1p (LSU-RS-rdy1? (LSU-RS1 (MA-LSU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (exist-uncommitted-LRM-before-p i (INST-ra i) MT))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))
```

```
; Similar to exist-uncommitted-LRM-INST-ra-if-LSU-RS0.
(defthm exist-uncommitted-LRM-INST-rb-if-LSU-RS0
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(LSU RS0))
                  (not (b1p (LSU-RS-rdy2? (LSU-RS0 (MA-LSU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (exist-uncommitted-LRM-before-p i (INST-rb i) MT))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Similar to exist-uncommitted-LRM-INST-ra-if-LSU-RS0.
(defthm exist-uncommitted-LRM-INST-rb-if-LSU-RS1
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(LSU RS1))
                  (not (b1p (LSU-RS-rdy2? (LSU-RS1 (MA-LSU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (exist-uncommitted-LRM-before-p i (INST-rb i) MT))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Similar to exist-uncommitted-LRM-INST-ra-if-LSU-RS0.
(defthm exist-uncommitted-LRM-INST-rc-if-LSU-RS0
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(LSU RS0))
                  (not (b1p (LSU-RS-rdy3? (LSU-RS0 (MA-LSU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (exist-uncommitted-LRM-before-p i (INST-rc i) MT))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Similar to exist-uncommitted-LRM-INST-ra-if-LSU-RS0.
(defthm exist-uncommitted-LRM-INST-rc-if-LSU-RS1
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(LSU RS1))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (LSU-RS-rdy3? (LSU-RS1 (MA-LSU MA)))))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (exist-uncommitted-LRM-before-p i (INST-rc i) MT))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Src field of a reservation station contains the tag to refer to
; the last register modifier of the operand register.
(defthm LSU-RS0-src-INST-tag-LRM-before
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(LSU RS0))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (LSU-RS-rdy1? (LSU-RS0 (MA-LSU MA)))))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
```

```
            (equal (LSU-RS-src1 (LSU-RS0 (MA-LSU MA)))
                   (INST-tag (LRM-before i (INST-ra i) MT))))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Similar to LSU-RS0-src-INST-tag-LRM-before.
(defthm LSU-RS1-src1-INST-tag-LRM-before
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(LSU RS1))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (LSU-RS-rdy1? (LSU-RS1 (MA-LSU MA)))))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (LSU-RS-src1 (LSU-RS1 (MA-LSU MA)))
                    (INST-tag (LRM-before i (INST-ra i) MT))))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Similar to LSU-RS0-src-INST-tag-LRM-before.
(defthm LSU-RS0-src2-INST-tag-LRM-before
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(LSU RS0))
                  (not (b1p (LSU-RS-rdy2? (LSU-RS0 (MA-LSU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (LSU-RS-src2 (LSU-RS0 (MA-LSU MA)))
                    (INST-tag (LRM-before i (INST-rb i) MT))))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Similar to LSU-RS0-src-INST-tag-LRM-before.
(defthm LSU-RS1-src2-INST-tag-LRM-before
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(LSU RS1))
                  (not (b1p (LSU-RS-rdy2? (LSU-RS1 (MA-LSU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (LSU-RS-src2 (LSU-RS1 (MA-LSU MA)))
                    (INST-tag (LRM-before i (INST-rb i) MT))))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Similar to LSU-RS0-src-INST-tag-LRM-before.
(defthm LSU-RS0-src3-INST-tag-LRM-before
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(LSU RS0))
                  (not (b1p (LSU-RS-rdy3? (LSU-RS0 (MA-LSU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (LSU-RS-src3 (LSU-RS0 (MA-LSU MA)))
                    (INST-tag (LRM-before i (INST-rc i) MT))))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

; Similar to LSU-RS0-src-INST-tag-LRM-before.
```

```
(defthm LSU-RS1-src3-INST-tag-LRM-before
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(LSU RS1))
                  (not (b1p (LSU-RS-rdy3? (LSU-RS1 (MA-LSU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (LSU-RS-src3 (LSU-RS1 (MA-LSU MA)))
                    (INST-tag (LRM-before i (INST-rc i) MT))))
  :hints (("goal" :in-theory (enable consistent-RS-p-def)
                  :use (:instance consistent-RS-entry-p-if-INST-in))))

(encapsulate nil
(local
(defthm CDB-val-INST-src-val1-if-CDB-ready-for-LSU-RS0
     (implies (and (equal (INST-stg i) '(LSU RS0))
                   (not (b1p (LSU-RS-rdy1? (LSU-RS0 (MA-LSU MA)))))
                   (b1p (CDB-ready-for?
                         (LSU-RS-src1 (LSU-RS0 (MA-LSU MA))) MA))
                   (not (b1p (INST-LSU-op? i)))
                   (not (b1p (inst-specultv? i)))
                   (not (b1p (INST-modified? i)))
                   (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (INST-p i) (INST-in i MT))
              (equal (equal (CDB-val MA) (INST-src-val1 i)) T))
  :hints (("goal" :in-theory (e/d (CDB-VAL-INST-DEST-VAL
                                   INST-cntlv
                                   INST-LSU?
                                   equal-b1p-converter
                                   decode logbit* rdb lift-b-ops
                                   INST-LSU-op?
                                   INST-src-val1)
                                  (INST-LSU-IF-LSU-STG-P))
                  :use (:instance INST-LSU-IF-LSU-STG-P)))))

(local
(defthm CDB-val-INST-src-val2-if-CDB-ready-for-LSU-RS0
     (implies (and (equal (INST-stg i) '(LSU RS0))
                   (not (b1p (LSU-RS-rdy2? (LSU-RS0 (MA-LSU MA)))))
                   (b1p (CDB-ready-for?
                         (LSU-RS-src2 (LSU-RS0 (MA-LSU MA))) MA))
                   (not (b1p (inst-specultv? i)))
                   (not (b1p (INST-modified? i)))
                   (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (INST-p i) (INST-in i MT))
              (equal (equal (CDB-val MA) (INST-src-val2 i)) T))
  :hints (("goal" :in-theory (e/d (CDB-VAL-INST-DEST-VAL
                                   INST-cntlv
                                   INST-LSU?
                                   equal-b1p-converter
                                   decode logbit* rdb lift-b-ops
                                   INST-LSU-op?
                                   INST-src-val2)
                                  (INST-LSU-IF-LSU-STG-P))
                  :use (:instance INST-LSU-IF-LSU-STG-P)))))

(local
(defthm CDB-val-INST-src-val3-if-CDB-ready-for-LSU-RS0
     (implies (and (equal (INST-stg i) '(LSU RS0))
```

```
                  (not (b1p (LSU-RS-rdy3? (LSU-RS0 (MA-LSU MA)))))
                  (b1p (CDB-ready-for?
                        (LSU-RS-src3 (LSU-RS0 (MA-LSU MA))) MA))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (equal (CDB-val MA) (INST-src-val3 i)) T))
  :hints (("goal" :in-theory (e/d (CDB-VAL-INST-DEST-VAL
                                   INST-cntlv
                                   INST-LSU?
                                   equal-b1p-converter
                                   decode logbit* rdb lift-b-ops
                                   INST-LSU-op?
                                   INST-src-val3)
                                  (INST-LSU-IF-LSU-STG-P))
                  :use (:instance INST-LSU-IF-LSU-STG-P)))))

; The instruction invariants are preserved for i that stays in LSU-RS0.
(defthm LSU-RS0-inst-inv-step-INST-from-LSU-RS0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg I) '(LSU RS0))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(LSU RS0)))
             (LSU-RS0-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable lift-b-ops LSU-RS0-inst-inv
                                     step-LSU
                                     step-LSU-RS0
                                     exception-relations))))
)

(encapsulate nil
(local
(defthm CDB-val-INST-src-val1-if-CDB-ready-for-LSU-RS1
     (implies (and (equal (INST-stg i) '(LSU RS1))
                  (not (b1p (LSU-RS-rdy1? (LSU-RS1 (MA-LSU MA)))))
                  (b1p (CDB-ready-for?
                        (LSU-RS-src1 (LSU-RS1 (MA-LSU MA))) MA))
                  (not (b1p (INST-LSU-op? i)))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (equal (CDB-val MA) (INST-src-val1 i)) T))
  :hints (("goal" :in-theory (e/d (CDB-VAL-INST-DEST-VAL
                                   INST-cntlv
                                   INST-LSU?
                                   equal-b1p-converter
                                   decode logbit* rdb lift-b-ops
                                   INST-LSU-op?
                                   INST-src-val1)
                                  (INST-LSU-IF-LSU-STG-P))
                  :use (:instance INST-LSU-IF-LSU-STG-P)))))
```

```
(local
(defthm CDB-val-INST-src-val2-if-CDB-ready-for-LSU-RS1
     (implies (and (equal (INST-stg i) '(LSU RS1))
                   (not (b1p (LSU-RS-rdy2? (LSU-RS1 (MA-LSU MA)))))
                   (b1p (CDB-ready-for?
                         (LSU-RS-src2 (LSU-RS1 (MA-LSU MA))) MA))
                   (not (b1p (inst-specultv? i)))
                   (not (b1p (INST-modified? i)))
                   (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (INST-p i) (INST-in i MT))
              (equal (equal (CDB-val MA) (INST-src-val2 i)) T))
  :hints (("goal" :in-theory (e/d (CDB-VAL-INST-DEST-VAL
                                   INST-cntlv
                                   INST-LSU?
                                   equal-b1p-converter
                                   decode logbit* rdb lift-b-ops
                                   INST-LSU-op?
                                   INST-src-val2)
                                  (INST-LSU-IF-LSU-STG-P))
                  :use (:instance INST-LSU-IF-LSU-STG-P)))))

(local
(defthm CDB-val-INST-src-val3-if-CDB-ready-for-LSU-RS1
     (implies (and (equal (INST-stg i) '(LSU RS1))
                   (not (b1p (LSU-RS-rdy3? (LSU-RS1 (MA-LSU MA)))))
                   (b1p (CDB-ready-for?
                         (LSU-RS-src3 (LSU-RS1 (MA-LSU MA))) MA))
                   (not (b1p (inst-specultv? i)))
                   (not (b1p (INST-modified? i)))
                   (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (INST-p i) (INST-in i MT))
              (equal (equal (CDB-val MA) (INST-src-val3 i)) T))
  :hints (("goal" :in-theory (e/d (CDB-VAL-INST-DEST-VAL
                                   INST-cntlv
                                   INST-LSU?
                                   equal-b1p-converter
                                   decode logbit* rdb lift-b-ops
                                   INST-LSU-op?
                                   INST-src-val3)
                                  (INST-LSU-IF-LSU-STG-P))
                  :use (:instance INST-LSU-IF-LSU-STG-P)))))

; The instruction invariants are preserved for i that stays in LSU-RS1.
(defthm LSU-RS1-inst-inv-step-INST-from-LSU-RS1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg I) '(LSU RS1))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(LSU RS1)))
             (LSU-RS1-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable lift-b-ops inst-inv-def
                                     step-LSU
                                     step-LSU-RS1
                                     exception-relations)
                  :cases ((b1p (LSU-RS-RDY1? (LSU-RS1 (MA-LSU MA)))))))))
```

```
)

; A landmark lemma.
; The instruction invariants are preserved for i whose next stage is
; LSU-RS0.
(defthm LSU-RS0-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(LSU RS0)))
             (LSU-RS0-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :use (:instance stages-reachable-to-LSU-RS0))))

; A landmark lemma.
; The instruction invariants are preserved for i whose next stage is
; LSU-RS1.
(defthm LSU-RS1-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(LSU RS1)))
             (LSU-RS1-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :use (:instance stages-reachable-to-LSU-RS1))))

(deflabel begin-LSU-issue-logic-lemmas)
; Instruction in LSU are issued in program order.  We need extra care
; so that this order is preserved by the LSU issuing hardware.

; When instruction in RS0 is ready to be issued, operand 1 is ready.
(defthm LSU-RS-rdy1-if-RS0-issue-ready
    (implies (b1p (LSU-RS0-issue-ready? LSU))
             (b1p (LSU-RS-rdy1? (LSU-RS0 LSU))))
  :hints (("goal" :in-theory (enable lift-b-ops LSU-RS0-issue-ready?
                                     equal-b1p-converter))))

; When instruction in RS0 is ready to be issued, operand 2 is ready.
(defthm LSU-RS-rdy2-if-RS0-issue-ready
    (implies (and (not (b1p (LSU-RS-op (LSU-RS0 LSU))))
                  (b1p (LSU-RS0-issue-ready? LSU)))
             (b1p (LSU-RS-rdy2? (LSU-RS0 LSU))))
  :hints (("goal" :in-theory (enable lift-b-ops LSU-RS0-issue-ready?
                                     equal-b1p-converter))))

; When instruction in RS0 is ready to be issued, operand 3 is ready.
(defthm LSU-RS-rdy3-if-RS0-issue-ready
    (implies (and (b1p (LSU-RS-ld-st? (LSU-RS0 LSU)))
                  (b1p (LSU-RS0-issue-ready? LSU)))
             (b1p (LSU-RS-rdy3? (LSU-RS0 LSU))))
  :hints (("goal" :in-theory (enable lift-b-ops LSU-RS0-issue-ready?
                                     equal-b1p-converter))))
```

```
; When instruction in RS1 is ready to be issued, operand 1 is ready.
(defthm LSU-RS-rdy1-if-RS1-issue-ready
    (implies (b1p (LSU-RS1-issue-ready? LSU))
             (b1p (LSU-RS-rdy1? (LSU-RS1 LSU))))
  :hints (("goal" :in-theory (enable lift-b-ops LSU-RS1-issue-ready?
                                     equal-b1p-converter))))

; When instruction in RS1 is ready to be issued, operand 2 is ready.
(defthm LSU-RS-rdy2-if-RS1-issue-ready
    (implies (and (not (b1p (LSU-RS-op (LSU-RS1 LSU))))
                  (b1p (LSU-RS1-issue-ready? LSU)))
             (b1p (LSU-RS-rdy2? (LSU-RS1 LSU))))
  :hints (("goal" :in-theory (enable lift-b-ops LSU-RS1-issue-ready?
                                     equal-b1p-converter))))

; When instruction in RS1 is ready to be issued, operand 3 is ready.
(defthm LSU-RS-rdy3-if-RS1-issue-ready
    (implies (and (b1p (LSU-RS-ld-st? (LSU-RS1 LSU)))
                  (b1p (LSU-RS1-issue-ready? LSU)))
             (b1p (LSU-RS-rdy3? (LSU-RS1 LSU))))
  :hints (("goal" :in-theory (enable lift-b-ops LSU-RS1-issue-ready?
                                     equal-b1p-converter))))

; Instructions in RS0 and RS1 cannot be ready to be issued simultaneously.
(defthm LSU-RS0-ISSUE-READY-LSU-RS1-ISSUE-READY-exclusive
    (implies (b1p (LSU-RS0-issue-ready? (MA-LSU MA)))
             (not (b1p (LSU-RS1-issue-ready? (MA-LSU MA)))))
  :hints (("goal" :in-theory (enable LSU-RS0-issue-ready? LSU-RS1-issue-ready?
                                     lift-b-ops))))

(deflabel end-LSU-issue-logic-lemmas)
(deftheory LSU-issue-logic-lemmas
    (set-difference-theories (universal-theory 'end-LSU-issue-logic-lemmas)
                             (universal-theory 'begin-LSU-issue-logic-lemmas)))
(in-theory (disable LSU-issue-logic-lemmas))

; The instruction invariants are preserved for instruction i that moves
; from LSU-RS0 to LSU-wbuf1.
(defthm LSU-wbuf1-inst-inv-step-INST-from-RS0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg I) '(LSU RS0))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(LSU wbuf1)))
             (LSU-WBUF1-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable lift-b-ops inst-inv-def
                                     step-LSU
                                     step-wbuf1
                                     LSU-issue-logic-lemmas
                                     issue-logic-def
                                     issued-write
                                     update-wbuf1
                                     INST-SRC-VAL1 INST-STORE-ADDR
                                     INST-SRC-VAL2 INST-SRC-VAL3
                                     INST-LSU-OP? INST-LD-ST?
                                     INST-SELECT-WBUF0?
```

```
                                    INST-store?
                                    decode rdb logbit*
                                    INST-cntlv
                                    exception-relations))))

; The instruction invariants are preserved for instruction i that moves
; from LSU-RS1 to LSU-wbuf1.
(defthm LSU-wbuf1-inst-inv-step-INST-from-RS1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg I) '(LSU RS1))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(LSU wbuf1)))
             (LSU-WBUF1-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable lift-b-ops inst-inv-def
                                    step-LSU
                                    step-wbuf1
                                    LSU-issue-logic-lemmas
                                    issue-logic-def
                                    issued-write
                                    update-wbuf1
                                    INST-SRC-VAL1 INST-STORE-ADDR
                                    INST-SRC-VAL2 INST-SRC-VAL3
                                    INST-SELECT-WBUF0?
                                    INST-LSU-OP? INST-LD-ST?
                                    INST-store?
                                    decode rdb logbit*
                                    INST-cntlv
                                    exception-relations))))

; The instruction invariants are preserved for instruction i that stays
; in wbuf1.
(defthm LSU-wbuf1-inst-inv-step-INST-from-wbuf1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg I) '(LSU wbuf1))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(LSU wbuf1)))
             (LSU-WBUF1-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable lift-b-ops inst-inv-def
                                    step-LSU step-wbuf1 issue-logic-def
                                    LSU-store-if-at-LSU-wbuf
                                    RELEASE-WBUF0?
                                    UPDATE-WBUF1
                                    exception-relations))))

; A landmark lemma.
; The instruction invariants are preserved for instruction i whose
; next stage is LSU-wbuf1.
(defthm LSU-wbuf1-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
```

```
                (MA-input-p sigs)
                (INST-p i)
                (INST-in i MT)
                (INST-inv i MA)
                (MT-no-jmp-exintr-before i MT MA sigs)
                (equal (INST-stg (step-INST I MT MA sigs))
                       '(LSU wbuf1)))
           (LSU-WBUF1-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :use (:instance stages-reachable-to-LSU-wbuf1))))

; The instruction invariants are preserved for instruction i that moves
; from RS0 to wbuf0.
(defthm LSU-wbuf0-inst-inv-step-INST-from-RS0
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (MA-input-p sigs)
                (INST-p i)
                (INST-in i MT)
                (INST-inv i MA)
                (MT-no-jmp-exintr-before i MT MA sigs)
                (equal (INST-stg I) '(LSU RS0))
                (equal (INST-stg (step-INST I MT MA sigs))
                       '(LSU wbuf0)))
           (LSU-WBUF0-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable lift-b-ops inst-inv-def
                                     step-LSU step-wbuf0
                                     LSU-issue-logic-lemmas
                                     issue-logic-def
                                     issued-write update-wbuf0
                                     wbuf1-output
                                     INST-SRC-VAL1 INST-STORE-ADDR
                                     INST-SRC-VAL2 INST-SRC-VAL3
                                     INST-SELECT-WBUF0?
                                     INST-LSU-OP? INST-LD-ST?
                                     INST-store?
                                     decode rdb logbit*
                                     INST-cntlv
                                     exception-relations))))

; The instruction invariants are preserved for instruction i that moves
; from RS1 to wbuf0.
(defthm LSU-wbuf0-inst-inv-step-INST-from-RS1
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (MA-input-p sigs)
                (INST-p i)
                (INST-in i MT)
                (INST-inv i MA)
                (MT-no-jmp-exintr-before i MT MA sigs)
                (equal (INST-stg I) '(LSU RS1))
                (equal (INST-stg (step-INST I MT MA sigs))
                       '(LSU wbuf0)))
           (LSU-WBUF0-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
    :hints (("goal" :in-theory (enable lift-b-ops inst-inv-def
                                     step-LSU step-wbuf0
                                     LSU-issue-logic-lemmas
                                     issue-logic-def
                                     issued-write update-wbuf0
                                     wbuf1-output
                                     INST-SRC-VAL1 INST-STORE-ADDR
                                     INST-SRC-VAL2 INST-SRC-VAL3
                                     INST-SELECT-WBUF0?
```

```
                                    INST-LSU-OP? INST-LD-ST?
                                    INST-store?
                                    decode rdb logbit*
                                    INST-cntlv
                                    exception-relations))))
(encapsulate nil
(local
 (defthm not-INST-excpt-detected-p-if-moved-from-wbuf1-to-wbuf0
     (implies (and (inv MT MA)
                   (equal (INST-stg i) '(LSU wbuf1))
                   (equal (INST-stg (step-INST i MT MA sigs))
                          '(LSU wbuf0))
                   (MAETT-p MT) (MA-state-p MA)
                   (INST-in i MT) (INST-p i)
                   (not (b1p (inst-specultv? i)))
                   (not (b1p (INST-modified? i))))
              (not (INST-excpt-detected-p (step-INST i MT MA sigs))))
   :hints (("goal" :in-theory (enable INST-excpt-detected-p
                                      INST-data-accs-error-detected-p
                                      INST-load-accs-error-detected-p
                                      INST-store-accs-error-detected-p
                                      exception-relations)))))

; The instruction invariants are preserved for instruction i that moves
; from wbuf1 to wbuf0.
(defthm LSU-wbuf0-inst-inv-step-INST-from-RS1-wbuf1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg I) '(LSU wbuf1))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(LSU wbuf0)))
             (LSU-WBUF0-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable lift-b-ops inst-inv-def
                                     step-LSU step-wbuf0 issue-logic-def
                                     LSU-store-if-at-LSU-wbuf
                                     RELEASE-WBUF0? wbuf1-output
                                     RELEASE-WBUF0-READY?
                                     issued-write UPDATE-WBUF0
                                     exception-relations))))
)

; The instruction invariants are preserved for instruction i that stays
; in wbuf0
(defthm LSU-wbuf0-inst-inv-step-INST-from-RS1-wbuf0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg I) '(LSU wbuf0))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(LSU wbuf0)))
             (LSU-WBUF0-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable lift-b-ops inst-inv-def
                                     step-LSU step-wbuf0 issue-logic-def
```

```
                                    LSU-store-if-at-LSU-wbuf
                                    RELEASE-WBUF0? wbuf1-output
                                    RELEASE-WBUF0-READY?
                                    issued-write UPDATE-WBUF0
                                    exception-relations))))

; A landmark lemma.
; The instruction invariants are preserved for instruction i whose next
; stage is wbuf0.
(defthm LSU-wbuf0-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(LSU wbuf0)))
             (LSU-WBUF0-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :use (:instance stages-reachable-to-LSU-wbuf0))))

(local
(defthm opcode-for-LSU-RS
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (or (EQUAL (INST-STG I) '(LSU RS0))
                      (EQUAL (INST-STG I) '(LSU RS1))))
             (or (equal (INST-opcode i) 3) (equal (INST-opcode i) 4)
                 (equal (INST-opcode i) 6) (equal (INST-opcode i) 7)))
  :hints (("goal" :in-theory (e/d (INST-LSU? INST-cntlv lift-b-ops
                                             equal-b1p-converter
                                             decode rdb logbit*)
                                  (INST-LSU-IF-LSU-STG-P))
                  :use (:instance INST-LSU-IF-LSU-STG-P)))
   :rule-classes nil))

(local
(defthm opcode-for-LSU-wbuf
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (or (EQUAL (INST-STG I) '(LSU wbuf0))
                      (EQUAL (INST-STG I) '(LSU wbuf1))
                      (EQUAL (INST-STG I) '(LSU wbuf0 lch))
                      (EQUAL (INST-STG I) '(LSU wbuf1 lch))))
             (or (equal (INST-opcode i) 4) (equal (INST-opcode i) 7)))
  :hints (("goal" :in-theory (e/d (INST-LSU? INST-cntlv lift-b-ops
                                             equal-b1p-converter
                                             INST-store? INST-ld-st?
                                             decode rdb logbit*)
                                  ())
                  :use (:instance LSU-store-if-at-LSU-wbuf)))
   :rule-classes nil))

(local
(defthm opcode-for-LSU-rbuf-lch
```

```
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (or (EQUAL (INST-STG I) '(LSU rbuf))
                      (EQUAL (INST-STG I) '(LSU lch))))
             (or (equal (INST-opcode i) 3) (equal (INST-opcode i) 6)))
  :hints (("goal" :in-theory (e/d (INST-LSU? INST-cntlv lift-b-ops
                                             equal-b1p-converter
                                             INST-load? INST-ld-st?
                                             decode rdb logbit*)
                                  ())
                  :use (:instance LSU-load-if-at-LSU-rbuf-lch)))
   :rule-classes nil))

; The instruction invariants are preserved for instruction i that moves
; from RS0 to rbuf.
(defthm LSU-rbuf-inst-inv-step-INST-from-LSU-RS0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i)
                  (INST-in i MT)
                  (MA-input-p sigs)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg i) '(LSU RS0))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(LSU rbuf)))
             (LSU-rbuf-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (e/d (LSU-rbuf-inst-inv
                                   issue-logic-def LSU-issue-logic-lemmas
                                   step-LSU step-rbuf
                                   INST-load-addr INST-src-val1
                                   INST-src-val2 INST-src-val3
                                   INST-LSU-OP?  INST-LD-ST?
                                   decode rdb logbit* INST-cntlv
                                   INST-load?
                                   lift-b-ops)
                                  ())
                  :use (:instance opcode-for-LSU-RS))))

; The instruction invariants are preserved for instruction i that moves
; from RS1 to rbuf.
(defthm LSU-rbuf-inst-inv-step-INST-from-LSU-RS1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i)
                  (INST-in i MT)
                  (MA-input-p sigs)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg i) '(LSU RS1))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(LSU rbuf)))
             (LSU-rbuf-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable LSU-rbuf-inst-inv
                                     issue-logic-def LSU-issue-logic-lemmas
                                     step-LSU step-rbuf
                                     INST-load-addr INST-src-val1
                                     INST-src-val2 INST-src-val3
                                     INST-LSU-OP?  INST-LD-ST?
```

```
                                    decode rdb logbit* INST-cntlv
                                    INST-load?
                                    lift-b-ops )
                  :use (:instance opcode-for-LSU-RS))))

; The instruction invariants are preserved for instruction i that stays
; in rbuf.
(defthm LSU-rbuf-inst-inv-step-INST-from-LSU-rbuf
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i)
                  (INST-in i MT)
                  (MA-input-p sigs)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg i) '(LSU rbuf))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(LSU rbuf)))
             (LSU-rbuf-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable lift-b-ops inst-inv-def
                                    step-LSU step-rbuf issue-logic-def
                                    LSU-load-if-at-LSU-rbuf-lch
                                    exception-relations))))

; A landmark lemma.
; The instruction invariants are preserved for instruction i whose next stage
; is rbuf.
(defthm LSU-rbuf-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i)
                  (INST-in i MT)
                  (MA-input-p sigs)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(LSU rbuf)))
             (LSU-rbuf-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :use (:instance stages-reachable-to-LSU-rbuf))))

; The write operation at wbuf0 is not ready to be released until it commits.
(defthm not-release-wbuf0-if-LSU-wbuf0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (or (equal (INST-stg i) '(LSU wbuf0))
                      (equal (INST-stg i) '(LSU wbuf0 lch))
                      (equal (INST-stg i) '(complete wbuf0))))
             (equal (release-wbuf0? (MA-LSU MA) sigs) 0))
  :hints (("goal" :in-theory (enable release-wbuf0? lift-b-ops
                                    equal-b1p-converter
                                    release-wbuf0-ready?))))

; If there is an unchecked write operation in wbuf0, a load instruction is
; not released.
(defthm not-release-rbuf-if-LSU-wbuf0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU wbuf0)))
             (equal (release-rbuf? (MA-LSU MA) MA sigs) 0))
  :hints (("goal" :in-theory (enable lift-b-ops CHECK-WBUF0?
```

```
                                    equal-b1p-converter
                                    release-rbuf?))))

; If there is an unchecked write operation in wbuf1, a load instruction is
; not released.
(defthm not-release-rbuf-if-LSU-wbuf1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU wbuf1)))
             (equal (release-rbuf? (MA-LSU MA) MA sigs) 0))
  :hints (("goal" :in-theory (enable lift-b-ops CHECK-WBUF1?
                                    equal-b1p-converter
                                    release-rbuf?))))

; The exception flags are set depending on whether the protection of the
; store memory address is violated or not.
(defthm INST-excpt-flags-step-inst-if-LSU-write-prohibited-at-wbuf0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU wbuf0))
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU wbuf0 lch))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (equal (INST-excpt-flags (step-INST i MT MA sigs))
                    (if (b1p (LSU-write-prohibited? (MA-LSU MA) (MA-mem MA)
                                                    (SRF-su (MA-SRF MA))))
                        6 0)))
  :hints (("goal" :in-theory (enable INST-excpt-flags
                                    INST-DATA-ACCS-ERROR-DETECTED-P
                                    INST-load-accs-error-detected-p
                                    INST-store-accs-error-detected-p
                                    LSU-WRITE-PROHIBITED?
                                    lift-b-ops
                                    write-error?)
                  :use (:instance opcode-for-LSU-wbuf))))

; Similar to INST-excpt-flags-step-inst-if-LSU-write-prohibited-at-wbuf0.
(defthm INST-excpt-flags-step-inst-if-LSU-write-prohibited-at-wbuf1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU wbuf1))
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU wbuf1 lch))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (equal (INST-excpt-flags (step-INST i MT MA sigs))
                    (if (b1p (LSU-write-prohibited? (MA-LSU MA) (MA-mem MA)
                                                    (SRF-su (MA-SRF MA))))
                        6 0)))
  :hints (("goal" :in-theory (enable INST-excpt-flags
                                    INST-DATA-ACCS-ERROR-DETECTED-P
                                    INST-load-accs-error-detected-p
                                    INST-store-accs-error-detected-p
                                    LSU-WRITE-PROHIBITED?
                                    check-wbuf0? check-wbuf1?
                                    lift-b-ops
                                    write-error?)
                  :use (:instance opcode-for-LSU-wbuf))))
```

```
; A landmark lemma.
; The instruction invariants are preserved for the instruction
; whose next address is LSU-wbuf-lch
(defthm LSU-wbuf0-lch-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(LSU wbuf0 lch)))
             (LSU-wbuf0-lch-inst-inv (step-INST i MT MA sigs)
                                     (MA-step MA sigs)))
  :hints (("goal" :use (:instance stages-reachable-to-LSU-wbuf0-lch)
                  :in-theory (enable LSU-wbuf0-lch-inst-inv
                                     lift-b-ops
                                     wbuf1-output step-wbuf0 update-wbuf0
                                     LSU-store-if-at-LSU-wbuf
                                     CHECK-WBUF0? issued-write
                                     step-LSU step-lsu-lch))))

; A landmark lemma.
; The instruction invariants are preserved for the instruction
; whose next stage is LSU-wbuf1-lch.
(defthm LSU-wbuf1-lch-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(LSU wbuf1 lch)))
             (LSU-wbuf1-lch-inst-inv (step-INST i MT MA sigs)
                                     (MA-step MA sigs)))
  :hints (("goal" :use (:instance stages-reachable-to-LSU-wbuf1-lch)
                  :in-theory (enable LSU-wbuf1-lch-inst-inv
                                     lift-b-ops
                                     wbuf1-output step-wbuf1 update-wbuf1
                                     LSU-store-if-at-LSU-wbuf
                                     CHECK-WBUF1? issued-write
                                     RELEASE-WBUF0?
                                     step-LSU step-lsu-lch))))

; From here, we prove lemma LSU-lch-inst-inv-step-INST.
; There are two major lemmas to prove.
; The first lemma to prove is
;   LSU-forward-wbuf-INST-dest-val
; This proves that load-forwarding is working fine.
; The second lemma
;   read-mem-INST-load-addr-INST-dest-val
; proves that memory access (possibly using load-bypassing) is working fine.

; If load instruction i violates the protection of the memory,
; the exception flags are set appropriately.
(defthm INST-excpt-flags-step-inst-if-LSU-read-prohibited-at-rbuf
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
```

```
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(LSU rbuf))
                (equal (INST-stg (step-INST i MT MA sigs))
                       '(LSU lch))
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i))))
           (equal (INST-excpt-flags (step-INST i MT MA sigs))
                  (if (b1p (LSU-read-prohibited? (MA-LSU MA) (MA-mem MA)
                                                 (SRF-su (MA-SRF MA))))
                      6 0)))
  :hints (("goal" :in-theory (enable INST-excpt-flags
                                     INST-DATA-ACCS-ERROR-DETECTED-P
                                     INST-load-accs-error-detected-p
                                     INST-store-accs-error-detected-p
                                     LSU-read-PROHIBITED?
                                     check-wbuf0? check-wbuf1?
                                     lift-b-ops
                                     read-error?)
                :use (:instance opcode-for-LSU-rbuf-lch))))

; INST-load-accs-error is detected when the released load instruction
; violates the memory access.
(defthm inst-load-accs-error-detected-p-if-lsu-read-prohibited
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU rbuf))
                  (b1p (release-rbuf? (MA-LSU MA) MA sigs))
                  (b1p (LSU-read-prohibited? (MA-LSU MA) (MA-mem MA)
                                             (SRF-su (MA-SRF MA))))
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (INST-load-accs-error-detected-p (step-INST i MT MA sigs)))
  :hints (("goal" :in-theory (enable LSU-read-prohibited? lift-b-ops
                                     read-error?
                                     INST-load-accs-error-detected-p)
                :use (:instance opcode-for-LSU-rbuf-lch))))

; From here, we start discussing the memory modifiers.
; The first goal is proving LSU-forward-wbuf-INST-dest-val.
; This lemma suggests that the load forwarding is working correctly.
(encapsulate nil
(local
(defthm not-no-inst-at-wbuf0-if-LSU-wbuf-valid-help-help
    (implies (uniq-INST-at-stgs-in-trace '((LSU WBUF0)
                                           (LSU WBUF0 LCH)
                                           (COMPLETE WBUF0)
                                           (COMMIT WBUF0)) trace)
             (not (no-inst-at-wbuf0-p trace)))))

(local
(defthm not-no-inst-at-wbuf0-if-LSU-wbuf-valid-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-INST-at-stgs '((LSU WBUF0)
                                       (LSU WBUF0 LCH)
                                       (COMPLETE WBUF0)
                                       (COMMIT WBUF0)) MT))
             (not (no-inst-at-wbuf0-p (MT-trace MT))))
  :hints (("goal" :in-theory (enable uniq-INST-at-stgs)))))

; A help lemma. The wbuf0 is valid, then no-inst-at-wbuf0-p is nil.
```

```
(defthm not-no-inst-at-wbuf0-if-LSU-wbuf-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
             (not (no-inst-at-wbuf0-p (MT-trace MT)))))
)

(encapsulate nil
(local
(defthm not-no-inst-at-wbuf1-if-LSU-wbuf-valid-help-help
    (implies (uniq-INST-at-stgs-in-trace '((LSU WBUF1)
                                           (LSU WBUF1 LCH)
                                           (COMPLETE WBUF1)
                                           (COMMIT WBUF1)) trace)
             (not (no-inst-at-wbuf1-p trace)))))

(local
(defthm not-no-inst-at-wbuf1-if-LSU-wbuf-valid-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-INST-at-stgs '((LSU WBUF1)
                                       (LSU WBUF1 LCH)
                                       (COMPLETE WBUF1)
                                       (COMMIT WBUF1)) MT))
             (not (no-inst-at-wbuf1-p (MT-trace MT))))
  :hints (("goal" :in-theory (enable uniq-INST-at-stgs)))))

; A help lemma. The wbuf1 is valid, then no-inst-at-wbuf1-p is nil.
(defthm not-no-inst-at-wbuf1-if-LSU-wbuf-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
             (not (no-inst-at-wbuf1-p (MT-trace MT)))))
)

(encapsulate nil
(local
(defthm exist-LMM-before-p-if-rbuf-wbuf0-help
     (implies (and (inv MT MA)
                   (subtrace-p trace MT)
                   (INST-listp trace)
                   (member-equal i trace) (INST-p i)
                   (MAETT-p MT) (MA-state-p MA)
                   (equal (INST-stg i) '(LSU rbuf))
                   (not (b1p (inst-specultv? i)))
                   (not (b1p (INST-modified? i)))
                   (in-order-wbuf0-rbuf-p trace)
                   (equal (wbuf-addr (LSU-wbuf0 (MA-LSU MA))) addr)
                   (not (no-inst-at-wbuf0-p trace)))
              (trace-exist-LMM-before-p i addr trace))
  :hints (("goal" :restrict ((LSU-WBUF0-ADDR-=-INST-STORE-ADDR
                              ((i (car trace)))))
                  :in-theory (enable wbuf0-stg-p)))))

; Suppose rbuf-wbuf0? flag is on, and the store instruction at wbuf0
; precedes the load instruction.  In that case, there is a memory
; modifier before the load instruction.
(defthm exist-LMM-before-p-if-rbuf-wbuf0
     (implies (and (inv MT MA)
                   (INST-in i MT) (INST-p i)
                   (MAETT-p MT) (MA-state-p MA)
                   (equal (INST-stg i) '(LSU rbuf))
```

```
                 (b1p (rbuf-wbuf0? (LSU-rbuf (MA-LSU MA))))
                 (equal (wbuf-addr (LSU-wbuf0 (MA-LSU MA))) addr)
                 (not (b1p (inst-specultv? i)))
                 (not (b1p (INST-modified? i))))
            (exist-LMM-before-p i addr MT))
  :hints (("goal" :in-theory (enable exist-LMM-before-p
                                     INST-in)
                  :restrict ((LSU-RBUF-VALID-IF-INST-IN ((i i)))))))
)

(encapsulate nil
(local
(defthm exist-LMM-before-p-if-rbuf-wbuf1-help
     (implies (and (inv MT MA)
                   (subtrace-p trace MT)
                   (INST-listp trace)
                   (member-equal i trace) (INST-p i)
                   (MAETT-p MT) (MA-state-p MA)
                   (equal (INST-stg i) '(LSU rbuf))
                   (not (b1p (inst-specultv? i)))
                   (not (b1p (INST-modified? i)))
                   (in-order-wbuf1-rbuf-p trace)
                   (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA))) addr)
                   (not (no-inst-at-wbuf1-p trace)))
              (trace-exist-LMM-before-p i addr trace))
  :hints (("goal" :restrict ((LSU-WBUF1-ADDR-=-INST-STORE-ADDR
                              ((i (car trace)))))
                  :in-theory (enable wbuf1-stg-p)))))

; Suppose rbuf-wbuf1? flag is 1, and the store instruction in wbuf1
; precedes the load instruction.  In that case, there is a memory
; modifier before the load instruction.
(defthm exist-LMM-before-p-if-rbuf-wbuf1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA))))
                  (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA))) addr)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (exist-LMM-before-p i addr MT))
  :hints (("goal" :in-theory (enable exist-LMM-before-p
                                     INST-in)
                  :restrict ((LSU-RBUF-VALID-IF-INST-IN ((i i)))))))
)

(encapsulate nil
(local
(defthm not-LSU-wbuf1-LMM-1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (exist-LMM-before-p i addr MT)
                  (not (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA))))))
             (not (equal (INST-stg (LMM-before i addr MT))
                         '(LSU wbuf1))))
  :hints (("goal" :use (:instance INST-in-order-p-rbuf-wbuf1
                                  (j (LMM-before i addr MT)))))))

(local
```

```
(defthm not-LSU-wbuf1-LMM-2
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (exist-LMM-before-p i addr MT)
                  (not (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA))) addr))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (not (equal (INST-stg (LMM-before i addr MT))
                         '(LSU wbuf1))))
  :hints (("goal" :restrict ((LSU-WBUF1-ADDR-=-INST-STORE-ADDR
                              ((i (LMM-before i addr MT)))))))))

(local
(defthm not-LSU-wbuf1-lch-LMM-1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (exist-LMM-before-p i addr MT)
                  (not (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA))))))
             (not (equal (INST-stg (LMM-before i addr MT))
                         '(LSU wbuf1 lch))))
  :hints (("goal" :use (:instance INST-in-order-p-rbuf-wbuf1
                                  (j (LMM-before i addr MT)))))))

(local
(defthm not-LSU-wbuf1-lch-LMM-2
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (exist-LMM-before-p i addr MT)
                  (not (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA))) addr))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (not (equal (INST-stg (LMM-before i addr MT))
                         '(LSU wbuf1 lch))))
  :hints (("goal" :restrict ((LSU-WBUF1-ADDR-=-INST-STORE-ADDR
                              ((i (LMM-before i addr MT)))))))))


(local
(defthm wbuf0-stg-p-LMM-before-if-execute-stg
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (b1p (rbuf-wbuf0? (LSU-rbuf (MA-LSU MA))))
                  (equal (wbuf-addr (LSU-wbuf0 (MA-LSU MA))) addr)
                  (or (not (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA)))))
                      (not (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA))) addr)))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (execute-stg-p
                   (INST-stg (LMM-before i addr MT))))
             (wbuf0-stg-p
              (INST-stg
               (LMM-before i addr MT))))
  :hints (("goal" :in-theory (enable execute-stg-p LSU-stg-p)))))
```

```
(local
(defthm not-complete-wbuf1-LMM-1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (exist-LMM-before-p i addr MT)
                  (not (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA))))))
             (not (equal (INST-stg (LMM-before i addr MT))
                         '(complete wbuf1))))
  :hints (("goal" :use (:instance INST-in-order-p-rbuf-wbuf1
                                  (j (LMM-before i addr MT)))))))

(local
(defthm not-complete-wbuf1-LMM-2
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (exist-LMM-before-p i addr MT)
                  (not (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA))) addr))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (not (equal (INST-stg (LMM-before i addr MT))
                         '(complete wbuf1))))
  :hints (("goal" :restrict ((LSU-WBUF1-ADDR-=-INST-STORE-ADDR
                              ((i (LMM-before i addr MT))))))))

(local
(defthm wbuf0-stg-p-LMM-before-if-complete
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (b1p (rbuf-wbuf0? (LSU-rbuf (MA-LSU MA))))
                  (equal (wbuf-addr (LSU-wbuf0 (MA-LSU MA))) addr)
                  (or (not (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA)))))
                      (not (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA))) addr)))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (complete-stg-p
                   (INST-stg (LMM-before i addr MT))))
             (wbuf0-stg-p
              (INST-stg
               (LMM-before i addr MT))))
    :hints (("goal" :in-theory (enable complete-stg-p)))))

(local
(defthm not-commit-wbuf1-LMM-1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (exist-LMM-before-p i addr MT)
                  (not (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA))))))
             (not (equal (INST-stg (LMM-before i addr MT))
                         '(commit wbuf1))))
  :hints (("goal" :use (:instance INST-in-order-p-rbuf-wbuf1
                                  (j (LMM-before i addr MT)))))))

(local
(defthm not-commit-wbuf1-LMM-2
```

```
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (exist-LMM-before-p i addr MT)
                  (not (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA))) addr))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (not (equal (INST-stg (LMM-before i addr MT))
                         '(commit wbuf1))))
  :hints (("goal" :restrict ((LSU-WBUF1-ADDR-=-INST-STORE-ADDR
                              ((i (LMM-before i addr MT)))))))))

(local
(defthm wbuf0-stg-p-LMM-before-if-commit
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (b1p (rbuf-wbuf0? (LSU-rbuf (MA-LSU MA))))
                  (equal (wbuf-addr (LSU-wbuf0 (MA-LSU MA))) addr)
                  (or (not (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA)))))
                      (not (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA))) addr)))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (commit-stg-p
                   (INST-stg (LMM-before i addr MT))))
             (wbuf0-stg-p
              (INST-stg
               (LMM-before i addr MT))))
  :hints (("goal" :in-theory (enable commit-stg-p)))))

(local
(defthm not-retire-stg-p-LMM-before-help
    (implies (and (wbuf-stg-p (INST-stg (inst-at-stgs stgs MT)))
                  (retire-stg-p (INST-stg (LMM-before i addr MT))))
             (not (equal (LMM-before i addr MT)
                         (inst-at-stgs stgs MT))))
  :hints (("goal" :in-theory (enable wbuf-stg-p retire-stg-p)))))

(local
(defthm not-retire-stg-p-LMM-before
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (b1p (rbuf-wbuf0? (LSU-rbuf (MA-LSU MA))))
                  (equal (wbuf-addr (LSU-wbuf0 (MA-LSU MA))) addr)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (not (retire-stg-p
                   (INST-stg (LMM-before i addr MT)))))
  :hints (("goal" :use ((:instance INST-in-order-p-retire-wbuf0
                                   (i (LMM-before i addr MT))
                                   (j (INST-at-stgs '((LSU wbuf0)
                                                      (LSU wbuf0 lch)
                                                      (complete wbuf0)
                                                      (commit wbuf0)) MT)))
                        (:instance INST-in-order-p-wbuf0-rbuf
                                   (j i)
                                   (i (INST-at-stgs '((LSU wbuf0)
                                                      (LSU wbuf0 lch)
```

```
                                              (complete wbuf0)
                                              (commit wbuf0)) MT)))
                        (:instance INST-IN-ORDER-P-INST-SPECULTV
                                   (i (INST-at-stgs '((LSU wbuf0)
                                                      (LSU wbuf0 lch)
                                                      (complete wbuf0)
                                                      (commit wbuf0)) MT))
                                   (j i))
                        (:instance INST-IN-ORDER-P-INST-MODIFIED
                                   (i (INST-at-stgs '((LSU wbuf0)
                                                      (LSU wbuf0 lch)
                                                      (complete wbuf0)
                                                      (commit wbuf0)) MT))
                                   (j i)))
                    :in-theory (e/d (LSU-wbuf-field-inst-at-lemmas)
                                    (INST-IN-ORDER-P-INST-SPECULTV
                                     INST-IN-ORDER-P-INST-MODIFIED))))))

; If rbuf-wbuf0? is 1, (i.e., the store instruction at wbuf0 precedes
; the load instruction), the store address and the load address match,
; and wbuf1 does not contain the memory modifier, then the last memory
; modifier before i is at a wbuf0 stage.  There are two conditions
; that the store instruction at wbuf1 is not a memory modifier.  The
; rbuf-wbuf1? is 0, or the load and store addresses don't match.
(defthm wbuf0-stg-p-LMM-before
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (b1p (rbuf-wbuf0? (LSU-rbuf (MA-LSU MA))))
                  (equal (wbuf-addr (LSU-wbuf0 (MA-LSU MA))) addr)
                  (or (not (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA)))))
                      (not (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA))) addr)))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (wbuf0-stg-p
              (INST-stg
               (LMM-before i addr MT))))
  :hints (("goal" :in-theory (e/d ()
                                  (INST-IN-ORDER-LMM-BEFORE
                                   INST-is-at-one-of-the-stages))
                  :use ((:instance INST-is-at-one-of-the-stages
                           (i (LMM-before i addr MT)))))))
)

(encapsulate  nil
(local
(defthm not-LSU-wbuf0-LMM-before-help
    (implies (and (wbuf1-stg-p (INST-stg (inst-at-stgs stgs MT)))
                  (wbuf0-stg-p (INST-stg
                                (LMM-before i addr MT))))
             (not (equal (LMM-before i addr MT)
                         (inst-at-stgs stgs MT))))
  :hints (("goal" :in-theory (enable wbuf1-stg-p wbuf0-stg-p)))))

(local
(defthm not-LSU-wbuf0-LMM-before
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA))))
```

```
                (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA))) addr)
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i))))
           (not (equal (INST-stg (LMM-before i addr MT))
                       '(LSU wbuf0))))
  :hints (("goal" :use ((:instance INST-in-order-p-wbuf0-wbuf1
                                   (i (LMM-before i addr MT))
                                   (j (INST-at-stgs '((LSU wbuf1)
                                                      (LSU wbuf1 lch)
                                                      (complete wbuf1)
                                                      (commit wbuf1)) MT)))
                        (:instance INST-in-order-p-wbuf1-rbuf
                                   (j i)
                                   (i (INST-at-stgs '((LSU wbuf1)
                                                      (LSU wbuf1 lch)
                                                      (complete wbuf1)
                                                      (commit wbuf1)) MT)))
                        (:instance INST-IN-ORDER-P-INST-SPECULTV
                                   (i (INST-at-stgs '((LSU wbuf1)
                                                      (LSU wbuf1 lch)
                                                      (complete wbuf1)
                                                      (commit wbuf1)) MT))
                                   (j i))
                        (:instance INST-IN-ORDER-P-INST-MODIFIED
                                   (i (INST-at-stgs '((LSU wbuf1)
                                                      (LSU wbuf1 lch)
                                                      (complete wbuf1)
                                                      (commit wbuf1)) MT))
                                   (j i)))
                :in-theory (disable INST-IN-ORDER-P-INST-MODIFIED
                                    INST-IN-ORDER-P-INST-SPECULTV)))))

(local
(defthm not-LSU-wbuf0-lch-LMM-before
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA)
                (equal (INST-stg i) '(LSU rbuf))
                (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA))))
                (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA))) addr)
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i))))
           (not (equal (INST-stg (LMM-before i addr MT))
                       '(LSU wbuf0 lch))))
    :hints (("goal" :use ((:instance INST-in-order-p-wbuf0-wbuf1
                                   (i (LMM-before i addr MT))
                                   (j (INST-at-stgs '((LSU wbuf1)
                                                      (LSU wbuf1 lch)
                                                      (complete wbuf1)
                                                      (commit wbuf1)) MT)))
                        (:instance INST-in-order-p-wbuf1-rbuf
                                   (j i)
                                   (i (INST-at-stgs '((LSU wbuf1)
                                                      (LSU wbuf1 lch)
                                                      (complete wbuf1)
                                                      (commit wbuf1)) MT)))
                        (:instance INST-IN-ORDER-P-INST-SPECULTV
                                   (i (INST-at-stgs '((LSU wbuf1)
                                                      (LSU wbuf1 lch)
                                                      (complete wbuf1)
                                                      (commit wbuf1)) MT))
                                   (j i))
```

```
                        (:instance INST-IN-ORDER-P-INST-MODIFIED
                                   (i (INST-at-stgs '((LSU wbuf1)
                                                      (LSU wbuf1 lch)
                                                      (complete wbuf1)
                                                      (commit wbuf1)) MT))
                                   (j i)))
                    :in-theory (disable INST-IN-ORDER-P-INST-MODIFIED
                                        INST-IN-ORDER-P-INST-SPECULTV)))))

(local
(defthm wbuf1-stg-p-LMM-before-if-execute-stg
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA))))
                  (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA))) addr)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (execute-stg-p
                   (INST-stg (LMM-before i addr MT))))
             (wbuf1-stg-p (INST-stg (LMM-before i addr MT))))
    :hints (("goal" :in-theory (enable execute-stg-p LSU-stg-p)))))

(local
(defthm not-complete-wbuf0-LMM-before
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA))))
                  (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA))) addr)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (not (equal (INST-stg (LMM-before i addr MT))
                         '(complete wbuf0))))
    :hints (("goal" :use ((:instance INST-in-order-p-wbuf0-wbuf1
                                   (i (LMM-before i addr MT))
                                   (j (INST-at-stgs '((LSU wbuf1)
                                                      (LSU wbuf1 lch)
                                                      (complete wbuf1)
                                                      (commit wbuf1)) MT)))
                        (:instance INST-in-order-p-wbuf1-rbuf
                                   (j i)
                                   (i (INST-at-stgs '((LSU wbuf1)
                                                      (LSU wbuf1 lch)
                                                      (complete wbuf1)
                                                      (commit wbuf1)) MT)))
                        (:instance INST-IN-ORDER-P-INST-SPECULTV
                                   (i (INST-at-stgs '((LSU wbuf1)
                                                      (LSU wbuf1 lch)
                                                      (complete wbuf1)
                                                      (commit wbuf1)) MT))
                                   (j i))
                        (:instance INST-IN-ORDER-P-INST-MODIFIED
                                   (i (INST-at-stgs '((LSU wbuf1)
                                                      (LSU wbuf1 lch)
                                                      (complete wbuf1)
                                                      (commit wbuf1)) MT))
                                   (j i)))
                    :in-theory (disable INST-IN-ORDER-P-INST-MODIFIED
                                        INST-IN-ORDER-P-INST-SPECULTV)))))
```

```
(local
(defthm wbuf1-stg-p-LMM-before-if-complete-stg
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA))))
                  (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA))) addr)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (complete-stg-p
                   (INST-stg (LMM-before i addr MT))))
             (wbuf1-stg-p (INST-stg (LMM-before i addr MT))))
  :hints (("goal" :in-theory (enable complete-stg-p)))))

(local
(defthm not-commit-wbuf0-LMM-before
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA))))
                  (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA))) addr)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (not (equal (INST-stg (LMM-before i addr MT))
                         '(commit wbuf0))))
    :hints (("goal" :use ((:instance INST-in-order-p-wbuf0-wbuf1
                                     (i (LMM-before i addr MT))
                                     (j (INST-at-stgs '((LSU wbuf1)
                                                        (LSU wbuf1 lch)
                                                        (complete wbuf1)
                                                        (commit wbuf1)) MT)))
                          (:instance INST-in-order-p-wbuf1-rbuf
                                     (j i)
                                     (i (INST-at-stgs '((LSU wbuf1)
                                                        (LSU wbuf1 lch)
                                                        (complete wbuf1)
                                                        (commit wbuf1)) MT)))
                          (:instance INST-IN-ORDER-P-INST-SPECULTV
                                     (i (INST-at-stgs '((LSU wbuf1)
                                                        (LSU wbuf1 lch)
                                                        (complete wbuf1)
                                                        (commit wbuf1)) MT))
                                     (j i))
                          (:instance INST-IN-ORDER-P-INST-MODIFIED
                                     (i (INST-at-stgs '((LSU wbuf1)
                                                        (LSU wbuf1 lch)
                                                        (complete wbuf1)
                                                        (commit wbuf1)) MT))
                                     (j i)))
                  :in-theory (disable INST-IN-ORDER-P-INST-MODIFIED
                                      INST-IN-ORDER-P-INST-SPECULTV)))))

(local
(defthm wbuf1-stg-p-LMM-before-if-commit-stg
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA))))
```

```
                (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA))) addr)
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (commit-stg-p
                 (INST-stg (LMM-before i addr MT))))
           (wbuf1-stg-p (INST-stg (LMM-before i addr MT))))
  :hints (("goal" :in-theory (enable commit-stg-p)))))

(local
(defthm not-retire-stg-LMM-before-help
    (implies (and (wbuf1-stg-p (INST-stg (inst-at-stgs stgs MT)))
                  (retire-stg-p (INST-stg
                                 (LMM-before i addr MT))))
             (not (equal (LMM-before i addr MT)
                         (inst-at-stgs stgs MT))))
  :hints (("goal" :in-theory (enable wbuf1-stg-p retire-stg-p)))))

(local
(defthm not-retire-stg-LMM-before
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (equal (INST-stg i) '(LSU rbuf))
                  (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA))))
                  (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA))) addr)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (not (retire-stg-p
                   (INST-stg (LMM-before i addr MT)))))
    :hints (("goal" :use ((:instance INST-in-order-p-retire-wbuf1
                                     (i (LMM-before i addr MT))
                                     (j (INST-at-stgs '((LSU wbuf1)
                                                        (LSU wbuf1 lch)
                                                        (complete wbuf1)
                                                        (commit wbuf1)) MT)))
                          (:instance INST-in-order-p-wbuf1-rbuf
                                     (j i)
                                     (i (INST-at-stgs '((LSU wbuf1)
                                                        (LSU wbuf1 lch)
                                                        (complete wbuf1)
                                                        (commit wbuf1)) MT)))
                          (:instance INST-IN-ORDER-P-INST-SPECULTV
                                     (i (INST-at-stgs '((LSU wbuf1)
                                                        (LSU wbuf1 lch)
                                                        (complete wbuf1)
                                                        (commit wbuf1)) MT))
                                     (j i))
                          (:instance INST-IN-ORDER-P-INST-MODIFIED
                                     (i (INST-at-stgs '((LSU wbuf1)
                                                        (LSU wbuf1 lch)
                                                        (complete wbuf1)
                                                        (commit wbuf1)) MT))
                                     (j i)))
                  :in-theory (disable INST-IN-ORDER-P-INST-MODIFIED
                                      INST-IN-ORDER-P-INST-SPECULTV)))))

; If rbuf-wbuf1? is on, and the load and store addresses match,
; then the last memory modifier before i is at wbuf1.
(defthm wbuf1-stg-p-LMM-before
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
```

```
                (equal (INST-stg i) '(LSU rbuf))
                (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA))))
                (equal (wbuf-addr (LSU-wbuf1 (MA-LSU MA))) addr)
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i))))
           (wbuf1-stg-p
            (INST-stg
             (LMM-before i addr MT))))
  :hints (("goal" :in-theory (e/d ()
                                  (INST-IN-ORDER-LMM-BEFORE
                                   INST-is-at-one-of-the-stages))
                :use ((:instance INST-is-at-one-of-the-stages
                         (i (LMM-before i addr MT)))))))
)

; If i is a memory modifier, the third operand value is stored in the
; memory at address addr.
(defthm read-mem-ISA-post-ISA-mem-modifier
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (mem-modifier-p addr i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (INST-exintr? i)))
                  (not (b1p (INST-excpt? i))))
             (equal (read-mem addr (ISA-mem (INST-post-ISA i)))
                    (INST-src-val3 i)))
  :hints (("goal" :in-theory (enable mem-modifier-p lift-b-ops
                                     INST-ld-st?  INST-excpt?
                                     INST-fetch-error? INST-cntlv
                                     INST-opcode INST-store-addr
                                     ISA-st ISA-sti
                                     INST-im INST-src-val3 INST-rc
                                     INST-rb INST-ra
                                     INST-data-access-error?
                                     INST-STORE-ERROR? INST-decode-error?
                                     DECODE-ILLEGAL-INST?
                                     ISA-step INST-store?))))

; An important lemma.
; This proves that the load-bypassing is working fine.
; The third source value of the last memory modifier before instruction i
; is the correct value read by instruction i from addr.
(defthm INST-src-val3-LMM-before
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (addr-p addr)
                  (exist-LMM-before-p i addr MT)
                  (not (retire-stg-p (INST-stg
                                      (LMM-before i addr MT))))
                  (execute-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (equal (INST-src-val3 (LMM-before i addr MT))
                    (read-mem addr (ISA-mem (INST-pre-ISA i)))))
  :hints (("goal" :use (:instance read-mem-LMM-before)
                  :in-theory (e/d (INST-exintr-INST-in-if-not-retired)
                                  (INST-POST-ISA-INST-IN)))))

; This lemmas says that the result of the load instruction i is
; the memory value at address (INST-load-addr i) in the pre-ISA of i.
(defthm read-mem-INST-load-addr-INST-pre-ISA
```

```
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (INST-load? i)))
             (equal (read-mem (INST-load-addr i) (ISA-mem (INST-pre-ISA i)))
                    (INST-dest-val i)))
  :hints (("goal" :In-theory (enable INST-load? INST-load-addr
                                     lift-b-ops INST-LSU? INST-ld-st?
                                     INST-cntlv INST-opcode
                                     INST-ld-dest-val INST-ld-im-dest-val
                                     INST-src-val1 INST-src-val2
                                     decode logbit* rdb
                                     INST-dest-val))))

; Field wbuf-val of write buffer 0 contains the result value of
; the load instruction i, if the rbuf-wbuf0? flag is on, and the load
; and store addresses match.
(defthm LSU-wbuf0-val-=-INST-dest-val
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU rbuf))
                  (MA-input-p sigs) (MA-state-p MA) (MAETT-p MT)
                  (b1p (rbuf-wbuf0? (LSU-rbuf (MA-LSU MA))))
                  (not (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA)))))
                  (equal (INST-load-addr i)
                         (wbuf-addr (LSU-wbuf0 (MA-LSU MA))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (equal (wbuf-val (LSU-wbuf0 (MA-LSU MA)))
                    (INST-dest-val i)))
  :hints (("goal" :in-theory (enable LSU-LOAD-IF-AT-LSU-RBUF-LCH)
                  :restrict
                  ((LSU-wbuf0-val-=-INST-src-val3
                    ((i (LMM-before i (INST-load-addr i) MT))))
                   (LSU-wbuf0-addr-=-INST-store-addr
                    ((i (LMM-before i (INST-load-addr i) MT)))))))))

; Again field wbuf-val of wbuf0 contains the result value for
; instruction i.  The difference between the previous lemma and
; this lemma is the reason of wbuf1 not intercepting.
; In the previous lemma, rbuf-wbuf1? was 0.  In this lemma, the load
; address does not match the store address of the write in wbuf1.
(defthm LSU-wbuf0-val-=-INST-dest-val-2
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU rbuf))
                  (MA-input-p sigs) (MA-state-p MA) (MAETT-p MT)
                  (b1p (rbuf-wbuf0? (LSU-rbuf (MA-LSU MA))))
                  (equal (INST-load-addr i)
                         (wbuf-addr (LSU-wbuf0 (MA-LSU MA))))
                  (not (equal (INST-load-addr i)
                              (wbuf-addr (LSU-wbuf1 (MA-LSU MA)))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (equal (wbuf-val (LSU-wbuf0 (MA-LSU MA)))
                    (INST-dest-val i)))
  :hints (("goal" :in-theory (enable LSU-LOAD-IF-AT-LSU-RBUF-LCH)
                  :restrict
                  ((LSU-wbuf0-val-=-INST-src-val3
                    ((i (LMM-before i (INST-load-addr i) MT))))
                   (LSU-wbuf0-addr-=-INST-store-addr
                    ((i (LMM-before i (INST-load-addr i) MT)))))))))
```

```
; Field wbuf-val of write buffer 1 contain the result value for load
; instruction i, when the load and store addresses are equal.
(defthm LSU-wbuf1-val-=-INST-dest-val
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU rbuf))
                  (MA-input-p sigs) (MA-state-p MA) (MAETT-p MT)
                  (b1p (rbuf-wbuf1? (LSU-rbuf (MA-LSU MA))))
                  (equal (INST-load-addr i)
                         (wbuf-addr (LSU-wbuf1 (MA-LSU MA))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (equal (wbuf-val (LSU-wbuf1 (MA-LSU MA)))
                    (INST-dest-val i)))
  :hints (("goal" :in-theory (enable LSU-LOAD-IF-AT-LSU-RBUF-LCH)
                  :restrict
                  ((LSU-wbuf1-val-=-INST-src-val3
                    ((i (LMM-before i (INST-load-addr i) MT))))
                   (LSU-wbuf1-addr-=-INST-store-addr
                    ((i (LMM-before i (INST-load-addr i) MT)))))))))

; A landmark lemma.
; If LSU-address-match? is 1, then the load forwarding value LSU-forward-wbuf
; is the correct result of the load instruction at the rbuf stage.
; This guarantees that the load-forwarding is working correctly.
; See the definition of LSU-forward-wbuf.
(defthm LSU-forward-wbuf-INST-dest-val
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU rbuf))
                  (MA-input-p sigs)
                  (b1p (release-rbuf? (MA-LSU MA) MA sigs))
                  (b1p (LSU-address-match? (MA-LSU MA)))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (LSU-forward-wbuf (MA-LSU MA))
                    (INST-dest-val i)))
  :hints (("goal" :in-theory (enable LSU-forward-wbuf lift-b-ops
                                     LSU-address-match?))))

;; From here we prove read-mem-INST-load-addr-INST-dest-val.
;; This lemma proves that the normal memory read possibly
;; with load bypassing is working correctly.

; A help lemma.
; Suppose instruction i and  j (which is (car trace)).  J precedes i
; in program order.  If there is no memory modifier between
; j and i, then the memory value at address addr in the pre-ISA state of j
; and  the pre-ISA state of i are identical.
(local
(defthm read-mem-when-no-mem-modifier-before
    (implies (and (inv MT MA)
                  (consp trace)
                  (subtrace-p trace MT)
                  (not (trace-exist-LMM-before-p i addr trace))
                  (member-equal i trace)
                  (syntaxp (not (and (consp i) (equal (car i) 'car))))
                  (INST-listp trace)
                  (INST-p i)
                  (addr-p addr)
```

```
                (MAETT-p MT) (MA-state-p MA))
           (equal (read-mem addr (ISA-mem (INST-pre-ISA (car trace))))
                  (read-mem addr (ISA-mem (INST-pre-ISA i)))))
  :hints ((when-found (INST-PRE-ISA (CAR (CDR TRACE)))
                      (:cases ((consp (cdr trace)))))
          ("goal" :induct t))))

(encapsulate nil
; A help lemma.
; There is no retired memory modifier following a non-retired instruction.
(local
(defthm not-INST-in-order-p-retired-non-retired
    (implies (and (inv MT MA)
                  (not (retire-stg-p (INST-stg i)))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (retire-stg-p (INST-stg j))
                  (INST-in-order-p i j MT)
                  (MAETT-p MT) (MA-state-p MA))
             (not (mem-modifier-p addr j)))
  :hints (("goal" :in-theory (enable INST-in-order-p-retired-store-non-retired
                                     mem-modifier-p)))))

(local
(defthm not-trace-exist-mem-modifier-cdr-if-no-active-induct
    (implies (and (inv MT MA)
                  (subtrace-after-p j trace MT)
                  (INST-p j) (INST-in j MT)
                  (INST-listp trace)
                  (not (retire-stg-p (INST-stg j)))
                  (not (trace-exist-non-retired-LMM-before-p i addr trace))
                  (MAETT-p MT) (MA-state-p MA))
             (not (trace-exist-LMM-before-p i addr trace)))
  :hints (("goal" :restrict ((not-INST-in-order-p-retired-non-retired
                              ((i j))))))
  :rule-classes nil))

; A help lemma.
; If trace-exist-non-retired-LMM-before-p and
; trace-exist-LMM-before-p are equivalent on (cdr trace)
; if (car trace) is retired.
(local
(defthm not-trace-exist-mem-modifier-cdr-if-no-active
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (INST-p j) (INST-listp trace)
                  (not (retire-stg-p (INST-stg (car trace))))
                  (not (trace-exist-non-retired-LMM-before-p i addr
                                                             (cdr trace)))
                  (MAETT-p MT) (MA-state-p MA))
             (not (trace-exist-LMM-before-p i addr (cdr trace))))
  :hints (("goal" :use (:instance
                        not-trace-exist-mem-modifier-cdr-if-no-active-induct
                        (j (car trace)) (trace (cdr trace)))))))

(local
(defthm read-mem-when-no-active-mem-modifier-before-induct
    (implies (and (inv MT MA)
                  (consp trace)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (member-equal i trace)
```

```
                (INST-p i) (INST-in i MT)
                (addr-p addr)
                (not (retire-stg-p (INST-stg i)))
                (not (trace-exist-non-retired-LMM-before-p i addr trace))
                (MAETT-p MT) (MA-state-p MA))
           (equal (read-mem addr
                            (trace-mem trace
                                       (ISA-mem (INST-pre-ISA (car trace)))))
                  (read-mem addr (ISA-mem (INST-pre-ISA i)))))
  :hints ((when-found (INST-PRE-ISA (CAR (CDR TRACE)))
                      (:cases ((consp (cdr trace)))))
          ("goal" :restrict ((read-mem-when-no-mem-modifier-before
                              ((i i))))))
  :rule-classes nil))

(local
(defthm read-mem-when-no-active-mem-modifier-before-help
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (retire-stg-p (INST-stg i)))
                  (not (exist-non-retired-LMM-before-p i addr MT))
                  (MAETT-p MT) (MA-state-p MA)
                  (addr-p addr))
             (equal (read-mem addr (MT-mem MT))
                    (read-mem addr (ISA-mem (INST-pre-ISA i)))))
  :hints (("goal" :in-theory (e/d (exist-non-retired-LMM-before-p
                                   INST-in MT-mem)
                                  (MT-MEM-=-MA-MEM))
                  :use (:instance
                        read-mem-when-no-active-mem-modifier-before-induct
                        (trace (MT-trace MT)))))
  :rule-classes nil))

; A critical lemma.
;
; If there is no non-retired memory modifier before i, the memory value in
; the current state is the ideal load value for instruction i.  This
; helps to show that the load bypassing is working correctly.
(defthm read-mem-when-no-active-mem-modifier-before
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (retire-stg-p (INST-stg i)))
                  (not (exist-non-retired-LMM-before-p i addr MT))
                  (addr-p addr)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (read-mem addr (MA-mem MA))
                    (read-mem addr (ISA-mem (INST-pre-ISA i)))))
  :hints (("goal" :use (:instance
                        read-mem-when-no-active-mem-modifier-before-help))))
)

(local
(defthm not-wbuf0-p-mem-modifier-if-LSU-address-match
    (implies (and (inv MT MA)
                  (not (b1p (LSU-address-match? (MA-LSU MA))))
                  (INST-in-order-p i j MT)
                  (equal (INST-stg j) '(LSU rbuf))
                  (wbuf0-stg-p (INST-stg i))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (inst-specultv? j)))
```

```
                (not (b1p (INST-modified? j))))
            (not (mem-modifier-p (INST-load-addr j) i)))
  :hints (("Goal" :in-theory (enable lift-b-ops LSU-address-match?
                                     mem-modifier-p
                                     inst-in-order-p-rbuf-wbuf0)
                  :cases ((b1p (inst-specultv? i)) (b1p (INST-modified? i)))
                  :restrict ((LSU-WBUF0-ADDR-=-INST-STORE-ADDR
                              ((i i)))))))

(local
(defthm not-wbuf1-p-mem-modifier-if-LSU-address-match
    (implies (and (inv MT MA)
                  (not (b1p (LSU-address-match? (MA-LSU MA))))
                  (INST-in-order-p i j MT)
                  (equal (INST-stg j) '(LSU rbuf))
                  (wbuf1-stg-p (INST-stg i))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (inst-specultv? j)))
                  (not (b1p (INST-modified? j))))
             (not (mem-modifier-p (INST-load-addr j) i)))
  :hints (("Goal" :in-theory (enable lift-b-ops LSU-address-match?
                                     mem-modifier-p
                                     inst-in-order-p-rbuf-wbuf1)
                  :cases ((b1p (inst-specultv? i)) (b1p (INST-modified? i)))
                  :restrict ((LSU-WBUF1-ADDR-=-INST-STORE-ADDR
                              ((i i)))))))

(local
(defthm not-mem-modifier-p-if-IU-MU-BU
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (or (BU-stg-p (INST-stg i)) (IU-stg-p (INST-stg i))
                      (MU-stg-p (INST-stg i)))
                  (INST-p i)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (not (mem-modifier-p addr i)))
  :hints (("goal" :in-theory (enable mem-modifier-p equal-b1p-converter
                                     INST-TYPE-EXCLUSIVE)
                  :use ((:instance INST-BU-IF-BU-STG-P)
                        (:instance INST-IU-IF-IU-STG-P)
                        (:instance INST-MU-IF-MU-STG-P))))))

(local
(defthm not-mem-modifier-p-if-rbuf-lch
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (or (equal (INST-stg i) '(LSU rbuf))
                      (equal (INST-stg i) '(LSU lch)))
                  (INST-p i)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (not (mem-modifier-p addr i)))
  :hints (("goal" :in-theory (enable mem-modifier-p
                                     LSU-LOAD-IF-AT-LSU-RBUF-LCH)))))

(local
(defthm not-mem-modifier-p-if-complete
```

```
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (equal (INST-stg i) '(complete))
                  (INST-p i)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (INST-decode-error-detected-p i))
                  (MAETT-p MT) (MA-state-p MA))
             (not (mem-modifier-p addr i)))
  :hints (("goal" :in-theory (enable mem-modifier-p
                                     not-INST-store-if-complete)))))
(local
(in-theory (disable not-mem-modifier-p-if-complete
                    not-mem-modifier-p-if-rbuf-lch)))


(encapsulate nil
(local
(defthm retired-stg-p-mem-modifier-if-not-LSU-address-match-case1
    (implies (and (inv MT MA)
                  (not (b1p (LSU-address-match? (MA-LSU MA))))
                  (INST-in-order-p i j MT)
                  (equal (INST-stg j) '(LSU rbuf))
                  (commit-stg-p (INST-stg i))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (inst-specultv? j)))
                  (not (b1p (INST-modified? j))))
             (not (mem-modifier-p (INST-load-addr j) i)))
  :hints (("Goal" :in-theory (enable commit-stg-p)))))

(local
(defthm retired-stg-p-mem-modifier-if-not-LSU-address-match-case2
    (implies (and (inv MT MA)
                  (not (b1p (LSU-address-match? (MA-LSU MA))))
                  (INST-in-order-p i j MT)
                  (equal (INST-stg j) '(LSU rbuf))
                  (complete-stg-p (INST-stg i))
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (inst-specultv? j)))
                  (not (b1p (INST-modified? j))))
             (not (mem-modifier-p (INST-load-addr j) i)))
  :hints (("goal" :in-theory (enable complete-stg-p
                                     lift-b-ops
                                     INST-excpt?
                                     not-mem-modifier-p-if-complete)
                  :cases ((b1p (inst-specultv? i))
                          (b1p (INST-modified? i))
                          (b1p (INST-fetch-error? i))
                          (b1p (INST-decode-error? i)))))))

(local
 (defthm retired-stg-p-mem-modifier-if-not-LSU-address-match-case3
    (implies (and (inv MT MA)
                  (not (b1p (LSU-address-match? (MA-LSU MA))))
                  (INST-in-order-p i j MT)
                  (equal (INST-stg j) '(LSU rbuf))
                  (execute-stg-p (INST-stg i))
```

```
                (INST-in i MT) (INST-p i)
                (INST-in j MT) (INST-p j)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (inst-specultv? j)))
                (not (b1p (INST-modified? j))))
           (not (mem-modifier-p (INST-load-addr j) i)))
  :hints (("goal" :in-theory (enable execute-stg-p LSU-stg-p
                                     not-mem-modifier-p-if-rbuf-lch
                                     INST-in-order-p-LSU-issued-RS)
                :cases ((b1p (inst-specultv? i))
                        (b1p (INST-modified? i)))))))

; Suppose instruction i is a memory modifier of the location where
; j is going to read.  If LSU-address-match? is 0, i must have already
; retired.
(defthm retired-stg-p-mem-modifier-if-not-LSU-address-match
    (implies (and (inv MT MA)
                (not (b1p (LSU-address-match? (MA-LSU MA))))
                (INST-in-order-p i j MT)
                (equal (INST-stg j) '(LSU rbuf))
                (not (retire-stg-p (INST-stg i)))
                (INST-in i MT) (INST-p i)
                (INST-in j MT) (INST-p j)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (inst-specultv? j)))
                (not (b1p (INST-modified? j))))
           (not (mem-modifier-p (INST-load-addr j) i)))
  :hints (("goal" :use ((:instance INST-is-at-one-of-the-stages (i i)))
                :in-theory (disable INST-is-at-one-of-the-stages))))
)

(encapsulate  nil
(local
(defthm not-exist-non-retired-LMM-before-p-if-not-address-match-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT)
                (equal (INST-stg i) '(LSU rbuf))
                (not (b1p (LSU-address-match? (MA-LSU MA))))
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (member-equal i trace)
                (INST-p i) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA))
           (not (trace-exist-non-retired-LMM-before-p i (INST-load-addr i)
                                                      trace)))
  :hints (("goal" :in-theory (enable exception-relations)))))

; If LSU-address-match? is 0, there is no non-retired memory modifier
; at the address specified by INST-load-addr.
(defthm not-exist-non-retired-LMM-before-p-if-not-address-match
    (implies (and (inv MT MA)
                (equal (INST-stg i) '(LSU rbuf))
                (not (b1p (LSU-address-match? (MA-LSU MA))))
                (not (b1p (inst-specultv? i)))
                (not (b1p (INST-modified? i)))
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA))
           (not (exist-non-retired-LMM-before-p i (INST-load-addr i) MT)))
  :hints (("goal" :in-theory (enable INST-in
                                     exist-non-retired-LMM-before-p))))
)
```

```
; A landmark lemma.
; If LSU-address-match? is 0, the value in the memory
; is, in fact, the destination value of instruction i.
; This guarantees that the load bypassing is working correctly.
(defthm read-mem-INST-load-addr-INST-dest-val
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU rbuf))
                  (not (b1p (LSU-address-match? (MA-LSU MA))))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (read-mem (INST-load-addr i) (MA-mem MA))
                    (INST-dest-val i)))
  :hints (("goal" :in-theory (enable LSU-LOAD-IF-AT-LSU-RBUF-LCH))))

; A landmark lemma
; The instruction invariants are preserved if i's next stage is LSU-lch.
; The proof has to consider the load-bypassing and load-forwarding.
; The proof uses two lemmas:
;   LSU-forward-wbuf-INST-dest-val
;   read-mem-INST-load-addr-INST-dest-val
(defthm LSU-lch-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(LSU lch)))
             (LSU-lch-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :use (:instance stages-reachable-to-LSU-lch)
                  :in-theory (enable LSU-lch-inst-inv MA-step lift-b-ops
                                     LSU-LOAD-IF-AT-LSU-RBUF-LCH
                                     CHECK-WBUF0? check-wbuf1?
                                     release-rbuf?
                                     step-LSU step-LSU-lch))))

; A landmark lemma
; Instruction invariants are preserved in the LSU.
(defthm LSU-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                  (LSU-stg-p (INST-stg (step-INST I MT MA sigs))))
             (LSU-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable LSU-stg-p LSU-inst-inv))))

;;;;;;;;;;;;Proof of execute-inst-robe-inv-step-INST;;;;;;;;;;;;;;;;;

; When an instruction is dispatched, the entry at the tail of the ROB
; records the instruction.
(defthm robe-receive-inst-MT-rob-tail
    (implies (and (b1p (dispatch-inst? MA))
                  (inv MT MA)
```

```
                (MAETT-p MT) (MA-state-p MA))
           (equal (robe-receive-INST? (MA-rob MA) (MT-rob-tail MT) MA) 1))
  :hints (("goal" :in-theory (enable robe-receive-INST?
                                     equal-b1p-converter))))

; If both instructions dispatch and commit take place this cycle,
; ROB-head and ROB-tail are not identical, i.e., the ROB is not empty
; nor full.
(defthm ROB-head-tail-not-equal-if-dispatch-and-commit
    (implies (and (inv MT MA)
                  (b1p (dispatch-inst? MA))
                  (b1p (commit-inst? MA))
                  (MAETT-p MT) (MA-state-p MA))
             (not (equal (MT-ROB-head MT) (MT-ROB-tail MT))))
  :hints (("goal" :in-theory (enable MA-def lift-b-ops))))

; The output of dispatch-pc from the dispatch logic is the program
; counter value for the dispatched instruction i.
(defthm dispatch-pc-INST-pc
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT))
             (equal (dispatch-pc MA) (INST-pc i)))
  :hints (("goal" :in-theory (enable dispatch-pc))))

; The output dispatch-dest-reg from the dispatch logic is the
; destination register for the dispatched instruction i.
(defthm dispatch-dest-reg-INST-dest-reg
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(DQ 0))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (not (INST-fetch-error-detected-p i))
                  (INST-writeback-p i))
             (equal (dispatch-dest-reg MA) (INST-dest-reg i)))
  :hints (("goal" :in-theory (enable dispatch-dest-reg
                                     DQ-OUT-DEST-REG
                                     INST-cntlv
                                     INST-opcode
                                     lift-b-ops
                                     INST-writeback-p
                                     INST-DEST-REG))))

(local
(defthm INST-cntlv-exunit-branch-0-if-decode-error
    (implies (and (INST-p i) (b1p (INST-decode-error? i)))
             (equal (logbit 3 (cntlv-exunit (INST-cntlv i))) 0))
  :hints (("goal" :in-theory (enable INST-cntlv INST-decode-error?
                                     decode-illegal-inst?
                                     bv-eqv-iff-equal
                                     equal-b1p-converter
                                     INST-opcode
                                     decode rdb logbit* lift-b-ops)))))

; The signal dispatch-excpt from a dispatch logic is the exception flags
; for the dispatched instruction.
(defthm dispatch-excpt-INST-excpt-flags
```

```
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(DQ 0))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (equal (dispatch-excpt MA) (INST-excpt-flags i)))
  :hints (("goal" :in-theory (enable dispatch-excpt))))

; An important lemma.
; Instruction-ROB invariants are preserved when instruction is dispatched
; from (DQ 0).
(defthm execute-inst-robe-inv-step-INST-from-DQ-0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg i) '(DQ 0))
                  (execute-stg-p (INST-stg (step-INST I MT MA sigs))))
     (execute-inst-robe-inv (step-INST i MT MA sigs)
                            (nth-robe (INST-tag (step-INST i MT MA sigs))
                                      (step-ROB MA sigs))))
  :hints (("goal" :in-theory (enable execute-inst-robe-inv lift-b-ops
                                     INST-pc
                                     INST-sync?
                                     INST-WB-SREG?
                                     INST-wb?
                                     INST-rfeh?
                                     INST-BU?
                                     word-addr-p
                                     step-robe)
                  :cases ((b1p (dispatch-inst? MA))))
          ("subgoal 1" :cases ((b1p (inst-fetch-error? i))))))

(encapsulate nil
(local
(defthm not-robe-receive-inst-INST-tag-help-help
    (implies (and (inv MT MA)
                  (consistent-rob-flg-p (MA-rob MA))
                  (consistent-robe-p (nth-robe rix (MA-rob MA))
                                     rix (MA-rob MA))
                  (b1p (robe-valid? (nth-robe rix (MA-rob MA))))
                  (not (b1p (ROB-full? (MA-rob MA))))
                  (MAETT-p MT) (MA-state-p MA))
             (not (equal rix (MT-rob-tail MT))))
  :hints (("goal" :in-theory (e/d (consistent-rob-flg-p
                                   lift-b-ops
                                   bv-eqv-iff-equal
                                   rob-full?
                                   consistent-robe-p)
                                  (CONSISTENT-ROBE-P-NTH-ROBE))))))

(local
(defthm not-robe-receive-inst-INST-tag-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (b1p (dispatch-inst? MA))
```

```
                (dispatched-p i)
                (not (committed-p i)))
           (not (equal (INST-tag i) (MT-rob-tail MT))))
  :hints (("goal" :in-theory (enable robe-receive-inst?
                                     equal-b1p-converter
                                     lift-b-ops)
                  :cases ((not (consistent-rob-flg-p (MA-rob MA)))
                          (not (consistent-robe-p (nth-robe (INST-tag i)
                                                            (MA-rob MA))
                                                  (INST-tag i) (MA-rob MA)))))
          ("subgoal 3" :in-theory (disable CONSISTENT-ROBE-P-NTH-ROBE)
                       :cases ((b1p (rob-full? (MA-rob MA)))))
          ("subgoal 2" :in-theory (enable inv consistent-MA-p
                                          consistent-rob-p)))
  :rule-classes nil))

; Suppose i is a dispatched but has not been committed.  The ROB entry
; to which i is assigned does not record another instruction.  In a sense,
; this proves that no resource conflict hazard occurs in the ROB.
(defthm not-robe-receive-inst-INST-tag
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (dispatched-p i)
                  (not (committed-p i)))
             (equal (robe-receive-inst? (MA-ROB MA) (INST-tag i) MA) 0))
  :hints (("goal" :in-theory (enable robe-receive-inst?
                                     equal-b1p-converter
                                     bv-eqv-iff-equal
                                     lift-b-ops)
                  :use (:instance not-robe-receive-inst-INST-tag-help))))
)

; Following proof may be tedious. You can skip to
;    complete-stg-step-INST-if-robe-receive-result
;
; A help lemma to prove
;  complete-stg-step-INST-if-robe-receive-result
; This is the case where i is current in the IU.
(defthm not-robe-receive-result-iff-complete-step-IU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (IU-stg-p (INST-stg i)))
             (iff (b1p (robe-receive-result? (MA-rob MA) (INST-tag i) MA))
                  (complete-stg-p (INST-stg (step-INST-IU i MA sigs)))))
  :hints (("goal" :in-theory (enable robe-receive-result? lift-b-ops
                                     CDB-def issue-logic-def
                                     IU-RS-field-INST-at-lemmas
                                     MU-field-INST-at-lemmas
                                     BU-RS-field-INST-at-lemmas
                                     LSU-field-INST-at-lemmas
                                     INST-at-stg-inst-stg-execute-2
                                     BU-output-def
                                     IU-output-def
                                     IU-stg-p
                                     step-INST-IU))))

; A help lemma to prove
;  complete-stg-step-INST-if-robe-receive-result
; This is the case where i is current in the MU.
(defthm not-robe-receive-result-iff-complete-step-MU
```

```
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (MU-stg-p (INST-stg i)))
             (iff (b1p (robe-receive-result? (MA-rob MA) (INST-tag i) MA))
                  (complete-stg-p (INST-stg (step-INST-MU i MA sigs)))))
  :hints (("goal" :in-theory (enable robe-receive-result? lift-b-ops
                                     CDB-def issue-logic-def
                                     IU-RS-field-INST-at-lemmas
                                     MU-field-INST-at-lemmas
                                     BU-RS-field-INST-at-lemmas
                                     LSU-field-INST-at-lemmas
                                     INST-at-stg-inst-stg-execute-2
                                     bv-eqv-iff-equal
                                     BU-output-def
                                     IU-output-def
                                     MU-stg-p
                                     step-INST-MU))))

; A help lemma to prove
;  complete-stg-step-INST-if-robe-receive-result
; This is the case where i is current in the BU.
(defthm not-robe-receive-result-iff-complete-step-BU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (BU-stg-p (INST-stg i)))
             (iff (b1p (robe-receive-result? (MA-rob MA) (INST-tag i) MA))
                  (complete-stg-p (INST-stg (step-INST-BU i MA sigs)))))
  :hints (("goal" :in-theory (enable robe-receive-result? lift-b-ops
                                     CDB-def issue-logic-def
                                     IU-RS-field-INST-at-lemmas
                                     MU-field-INST-at-lemmas
                                     BU-RS-field-INST-at-lemmas
                                     LSU-field-INST-at-lemmas
                                     BU-output-def IU-output-def
                                     INST-at-stg-inst-stg-execute-2
                                     BU-stg-p step-INST-BU))))

; WARNING!!!
; Proof of a help lemma for complete-stg-step-INST-if-robe-receive-result
; is tedious and requires more lemmas.
; You can skip to not-robe-receive-result-iff-complete-step-LSU

; Instruction at reservation station LSU-RS0 does not complete its execution
; this cycle.
(defthm not-complete-step-inst-LSU-RS0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (equal (INST-stg i) '(LSU RS0)))
             (not (complete-stg-p
                   (INST-stg (step-INST-LSU-RS0 i MA sigs)))))
  :hints (("goal" :in-theory (enable step-INST-LSU-RS0))))

; The ROB entry which holds the instruction at LSU-RS0 does not get the
; result this cycle.
(defthm not-robe-receive-result-if-LSU-RS0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (equal (INST-stg i) '(LSU RS0)))
```

```
            (equal (robe-receive-result? (MA-rob MA) (INST-tag i) MA) 0))
  :hints (("goal" :in-theory (enable robe-receive-result? lift-b-ops
                                     equal-b1p-converter
                                     CDB-def issue-logic-def
                                     IU-RS-field-INST-at-lemmas
                                     MU-field-INST-at-lemmas
                                     BU-RS-field-INST-at-lemmas
                                     LSU-field-INST-at-lemmas
                                     BU-output-def
                                     IU-output-def
                                     INST-at-stg-inst-stg-execute-2))))

; Instruction at reservation station LSU-RS1 does not complete its execution
; this cycle.
(defthm not-robe-receive-result-iff-complete-step-LSU-RS1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (equal (INST-stg i) '(LSU RS1)))
             (not (complete-stg-p
                   (INST-stg (step-INST-LSU-RS1 i MA sigs)))))
  :hints (("goal" :in-theory (enable step-INST-LSU-RS1))))

; The ROB entry which holds the instruction at LSU-RS1 does not get the
; result this cycle.
(defthm not-robe-receive-result-if-LSU-RS1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (equal (INST-stg i) '(LSU RS1)))
             (equal (robe-receive-result? (MA-rob MA) (INST-tag i) MA) 0))
  :hints (("goal" :in-theory (enable robe-receive-result? lift-b-ops
                                     equal-b1p-converter
                                     CDB-def issue-logic-def
                                     IU-RS-field-INST-at-lemmas
                                     MU-field-INST-at-lemmas
                                     BU-RS-field-INST-at-lemmas
                                     LSU-field-INST-at-lemmas
                                     BU-output-def
                                     IU-output-def
                                     step-INST-LSU-RS1
                                     INST-at-stg-inst-stg-execute-2))))

; Instruction at stage LSU-wbuf0 does not complete its execution
; this cycle.
(defthm not-complete-stg-p-step-INST-LSU-wbuf0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (equal (INST-stg i) '(LSU wbuf0)))
             (not (complete-stg-p (INST-stg (step-INST-LSU-wbuf0 i MA)))))
  :hints (("goal" :in-theory (enable step-INST-LSU-wbuf0))))

; The ROB entry which holds instruction at LSU-wbuf0 does not get the
; result this cycle.
(defthm not-robe-receive-result-if-LSU-wbuf0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (equal (INST-stg i) '(LSU wbuf0)))
             (equal (robe-receive-result? (MA-rob MA) (INST-tag i) MA) 0))
  :hints (("goal" :in-theory (enable robe-receive-result? lift-b-ops
```

```
                                    equal-b1p-converter
                                    CDB-def issue-logic-def
                                    IU-RS-field-INST-at-lemmas
                                    MU-field-INST-at-lemmas
                                    BU-RS-field-INST-at-lemmas
                                    LSU-field-INST-at-lemmas
                                    BU-output-def
                                    IU-output-def
                                    step-INST-LSU-wbuf0
                                    INST-at-stg-inst-stg-execute-2))))

; Instruction at stage LSU-wbuf1 does not complete its execution
; this cycle.
(defthm not-complete-stg-step-INST-LSU-wbuf1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (equal (INST-stg i) '(LSU wbuf1)))
             (not (complete-stg-p (INST-stg (step-INST-LSU-wbuf1 i MA sigs)))))
  :hints (("goal" :in-theory (enable step-INST-LSU-wbuf1))))

; The ROB entry which holds instruction at LSU-wbuf1 does not get the
; result this cycle.
(defthm not-robe-receive-result-if-LSU-wbuf1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (equal (INST-stg i) '(LSU wbuf1)))
             (equal (robe-receive-result? (MA-rob MA) (INST-tag i) MA) 0))
  :hints (("goal" :in-theory (enable robe-receive-result? lift-b-ops
                                    equal-b1p-converter
                                    CDB-def issue-logic-def
                                    IU-RS-field-INST-at-lemmas
                                    MU-field-INST-at-lemmas
                                    BU-RS-field-INST-at-lemmas
                                    LSU-field-INST-at-lemmas
                                    BU-output-def
                                    IU-output-def
                                    step-INST-LSU-RS1
                                    INST-at-stg-inst-stg-execute-2))))

; The instruction at stage LSU-rbuf does not complete its execution
; this cycle.
(defthm not-complete-stg-p-step-INST-LSU-rbuf
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (equal (INST-stg i) '(LSU rbuf)))
             (not (complete-stg-p (INST-stg (step-INST-LSU-rbuf i MA sigs)))))
  :hints (("goal" :in-theory (enable step-INST-LSU-rbuf))))

; The ROB entry which holds instruction at LSU-rbuf does not get the
; result this cycle.
(defthm not-robe-receive-result-iff-complete-step-LSU-rbuf
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (equal (INST-stg i) '(LSU rbuf)))
             (equal (robe-receive-result? (MA-rob MA) (INST-tag i) MA) 0))
  :hints (("goal" :in-theory (enable robe-receive-result? lift-b-ops
                                    equal-b1p-converter
                                    CDB-def issue-logic-def
```

```
                                    IU-RS-field-INST-at-lemmas
                                    MU-field-INST-at-lemmas
                                    BU-RS-field-INST-at-lemmas
                                    LSU-field-INST-at-lemmas
                                    BU-output-def
                                    IU-output-def
                                    step-INST-LSU-RS1
                                    INST-at-stg-inst-stg-execute-2))))

; The instruction at stage LSU-wbuf0-lch complete its execution
; this cycle.
(defthm complete-stg-p-step-INST-LSU-wbuf0-lch
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (equal (INST-stg i) '(LSU wbuf0 lch)))
             (complete-stg-p (INST-stg (step-INST-LSU-wbuf0-lch i))))
  :hints (("goal" :in-theory (enable step-INST-LSU-wbuf0-lch))))

(local
(defthm not-uniq-inst-at-stgs-if-inst-in
    (implies (and (inv MT MA)
                  (INST-p I) (INST-in i MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (member-equal (INST-stg i)
                                '((LSU LCH) (LSU WBUF0 LCH)
                                  (LSU WBUF1 LCH))))
             (equal (INST-at-stgs '((LSU LCH) (LSU WBUF0 LCH)
                                    (LSU WBUF1 LCH))
                                  MT)
                    i))
  :hints (("goal" :cases ((UNIQ-INST-AT-STGS '((LSU LCH)
                                               (LSU WBUF0 LCH)
                                               (LSU WBUF1 LCH))
                                             MT)
                          (NO-INST-AT-STGS '((LSU LCH)
                                             (LSU WBUF0 LCH)
                                             (LSU WBUF1 LCH))
                                           MT))
                  :in-theory (enable NOT-NO-INST-AT-STGS-INST-STG-IF-INST-IN
                                     INST-at-stgs-if-INST-in)))))

; The ROB entry holding the instruction at stage LSU-wbuf0-lch
; gets its result this cycle.
(defthm not-robe-receive-result-iff-complete-step-LSU-wbuf0-lch
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (equal (INST-stg i) '(LSU wbuf0 lch)))
             (equal (robe-receive-result? (MA-rob MA) (INST-tag i) MA) 1))
  :hints (("goal" :in-theory (enable robe-receive-result? lift-b-ops
                                     equal-b1p-converter
                                     CDB-def issue-logic-def
                                     IU-RS-field-INST-at-lemmas
                                     MU-field-INST-at-lemmas
                                     BU-RS-field-INST-at-lemmas
                                     LSU-field-INST-at-lemmas
                                     strong-inst-at-stg-theory
                                     BU-output-def
                                     IU-output-def
                                     INST-at-stg-inst-stg-execute-2))))
```

```
; The instruction at stage LSU-wbuf1-lch completes its execution
; this cycle.
(defthm complete-stg-p-step-INST-LSU-wbuf1-lch
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (equal (INST-stg i) '(LSU wbuf1 lch)))
             (complete-stg-p
              (INST-stg (step-INST-LSU-wbuf1-lch i MA sigs))))
  :hints (("goal" :in-theory (enable step-INST-LSU-wbuf1-lch))))

; The ROB entry holding the instruction at stage LSU-wbuf1-lch
; gets its result this cycle.
(defthm robe-receive-result-if-LSU-wbuf1-lch
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (equal (INST-stg i) '(LSU wbuf1 lch)))
             (equal (robe-receive-result? (MA-rob MA) (INST-tag i) MA) 1))
  :hints (("goal" :in-theory (enable robe-receive-result? lift-b-ops
                                     equal-b1p-converter
                                     CDB-def issue-logic-def
                                     IU-RS-field-INST-at-lemmas
                                     MU-field-INST-at-lemmas
                                     BU-RS-field-INST-at-lemmas
                                     LSU-field-INST-at-lemmas
                                     strong-inst-at-stg-theory
                                     BU-output-def
                                     IU-output-def
                                     INST-at-stg-inst-stg-execute-2))))

; Instruction at stage LSU-lch complete its execution
; this cycle.
(defthm not-robe-receive-result-iff-complete-step-LSU-lch
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (equal (INST-stg i) '(LSU lch)))
             (complete-stg-p (INST-stg (step-INST-LSU-lch i))))
  :hints (("goal" :in-theory (enable step-INST-LSU-lch))))

; The ROB entry holding the instruction at stage LSU-lch
; gets its result this cycle.
(defthm robe-receive-result-if-execute-LSU-lch
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (equal (INST-stg i) '(LSU lch)))
             (equal (robe-receive-result? (MA-rob MA) (INST-tag i) MA) 1))
  :hints (("goal" :in-theory (enable robe-receive-result? lift-b-ops
                                     equal-b1p-converter
                                     CDB-def issue-logic-def
                                     IU-RS-field-INST-at-lemmas
                                     MU-field-INST-at-lemmas
                                     BU-RS-field-INST-at-lemmas
                                     LSU-field-INST-at-lemmas
                                     strong-inst-at-stg-theory
                                     BU-output-def
                                     IU-output-def
                                     INST-at-stg-inst-stg-execute-2))))

; A help lemma to prove
```

```
;  complete-stg-step-INST-if-robe-receive-result
; This is the case where i is currently in the LSU.
(defthm not-robe-receive-result-iff-complete-step-LSU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (LSU-stg-p (INST-stg i)))
             (iff (b1p (robe-receive-result? (MA-rob MA) (INST-tag i) MA))
                  (complete-stg-p (INST-stg (step-INST-LSU i MA sigs)))))
  :hints (("goal" :in-theory (enable LSU-stg-p step-INST-LSU))))

; Suppose i is an instruction in an execution stage.
; The ROB entry assigned to i gets its result iff i completes this cycle.
(defthm complete-stg-step-INST-if-robe-receive-result
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (INST-p I)
                  (MAETT-p MT) (MA-state-p MA)
                  (execute-stg-p (INST-stg i)))
             (iff (complete-stg-p (INST-stg (step-INST i MT MA sigs)))
                  (b1p (robe-receive-result? (MA-rob MA) (INST-tag i) MA))))
  :hints (("goal" :in-theory (enable step-INST-opener lift-b-ops
                                     execute-stg-p
                                     step-INST-execute)))
  :rule-classes
  ((:rewrite :corollary
     (implies (and (inv MT MA)
                   (INST-in i MT) (INST-p i)
                   (MAETT-p MT) (MA-state-p MA)
                   (execute-stg-p (INST-stg i))
                   (b1p (robe-receive-result? (MA-rob MA) (INST-tag i) MA)))
              (complete-stg-p (INST-stg (step-INST i MT MA sigs)))))
   (:rewrite :corollary
     (implies (and (inv MT MA)
                   (INST-in i MT) (INST-p i)
                   (MAETT-p MT) (MA-state-p MA)
                   (execute-stg-p (INST-stg i))
                   (execute-stg-p (INST-stg i))
                   (not (b1p (robe-receive-result? (MA-rob MA)
                                                   (INST-tag i) MA))))
              (not (complete-stg-p (INST-stg (step-INST i MT MA sigs))))))))

(encapsulate nil
(local
(defthm not-CDB-ready-for-INST-tag-if-complete-help
    (implies (and (inv MT MA)
                  (complete-stg-p (INST-stg i))
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (CDB-ready? MA)))
             (not (equal (CDB-tag MA) (INST-tag i))))
  :hints (("goal" :in-theory (enable CDB-tag CDB-ready? lift-b-ops
                                     IU-output-def BU-output-def
                                     LSU-output-def MU-output-def
                                     CDB-def issue-logic-def
                                     LSU-field-lemmas MU-field-lemmas
                                     BU-RS-field-lemmas IU-RS-field-lemmas)))))

; The CDB is not ready for a completed instruction. (a kind of trivial fact.)
(defthm not-CDB-ready-for-INST-tag-if-complete
    (implies (and (inv MT MA)
                  (complete-stg-p (INST-stg i))
```

```
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA))
           (equal (CDB-ready-for? (INST-tag i) MA) 0))
  :hints (("goal" :in-theory (enable CDB-ready-for? lift-b-ops
                                     bv-eqv-iff-equal equal-b1p-converter))))
)

; If instruction i is in complete-stg, the ROB entry to which i is assigned
; does not receive another instruction.
(defthm not-robe-receive-result-if-complete-stg
    (implies (and (inv MT MA)
                  (complete-stg-p (INST-stg i))
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (robe-receive-result? (MA-rob MA) (INST-tag i) MA) 0))
  :hints (("goal" :in-theory (enable robe-receive-result?
                                     lift-b-ops equal-b1p-converter))))

; An important lemma.
; Instruction invariants for the ROB are preserved for instruction i if
; i stays in an execution stage.
(defthm execute-inst-robe-inv-step-INST-from-execute-stg
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (execute-stg-p (INST-stg i))
                  (execute-stg-p (INST-stg (step-INST I MT MA sigs))))
     (execute-inst-robe-inv (step-INST i MT MA sigs)
                            (nth-robe (INST-tag i) (step-ROB MA sigs))))
  :hints (("goal" :in-theory (e/d (lift-b-ops
                                   step-robe
                                   INST-pc
                                   exception-relations
                                   INST-excpt-detected-p
                                   execute-inst-robe-inv)
                                  (ROBE-BRANCH-INST-BRANCH-2)))))

; A landmark lemma
; The instruction-ROB invariants are preserved for instruction i, if
; i is in execute-stg in the next cycle.
; The proof is divided into the proofs of
;   execute-inst-robe-inv-step-INST-from-DQ-0
;   execute-inst-robe-inv-step-INST-from-execute-stg
(defthm execute-inst-robe-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (execute-stg-p (INST-stg (step-INST I MT MA sigs))))
     (execute-inst-robe-inv (step-INST i MT MA sigs)
                            (nth-robe (INST-tag (step-INST i MT MA sigs))
                                      (step-ROB MA sigs))))
  :hints (("goal" :use (:instance stages-reachable-to-execute)
                  :in-theory (disable nth-robe-step-rob))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; End of the proof of  execute-inst-robe-inv-step-INST
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; A landmark lemma execute-inst-inv-step-INST.
; The instruction invariants preserved for i if i is in execute-stg in
; the next cycle.
(defthm execute-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (execute-stg-p (INST-stg (step-INST I MT MA sigs))))
             (execute-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (e/d (execute-inst-inv execute-stg-p)
                                  (nth-robe-step-rob)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Proof of complete-inst-inv
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; The proof of complete-inst-inv is divided into
;  complete-normal-inst-inv-step-INST
;  complete-wbuf0-inst-inv-step-INST
;  complete-wbuf1-inst-inv-step-INST
; Before proving these lemmas, we also prove
;   complete-inst-robe-inv-step-INST,
; which is used in the proof of all three lemmas above.
(encapsulate nil
(local
 (defthm not-dispatch-no-unit-if-INST-bu
     (implies (and (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (inst-p i) (INST-in i MT)
                   (not (b1p (inst-specultv? i)))
                   (not (b1p (INST-modified? i)))
                   (equal (INST-stg i) '(DQ 0))
                   (not (b1p (INST-fetch-error? i)))
                   (b1p (INST-bu? i)))
              (equal (dispatch-no-unit? MA) 0))
 :hints (("goal" :in-theory (enable dispatch-no-unit? lift-b-ops
                                    equal-b1p-converter
                                    INST-bu? INST-cntlv
                                    INST-DECODE-ERROR-DETECTED-P
                                    decode logbit* rdb
                                    INST-excpt-detected-p
                                    DQ-READY-NO-UNIT?)))))

(local
 (defthm not-dispatch-no-unit-if-INST-write-back
     (implies (and (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (inst-p i) (INST-in i MT)
                   (equal (INST-stg i) '(DQ 0))
                   (not (b1p (inst-specultv? i)))
                   (not (b1p (INST-modified? i)))
                   (not (b1p (INST-fetch-error? i)))
                   (not (b1p (INST-decode-error? i)))
```

```
                (INST-writeback-p i))
           (equal (dispatch-no-unit? MA) 0))
   :hints (("goal" :in-theory (enable INST-writeback-p dispatch-no-unit?
                                      equal-b1p-converter
                                      DQ-ready-no-unit?
                                      INST-cntlv
                                      INST-excpt-detected-p
                                      INST-opcode
                                      lift-b-ops)))))

; The instruction ROB invariants is preserved for instruction i,
; when it advances from (DQ 0) to complete-stg.
(defthm complete-inst-robe-inv-step-INST-from-DQ0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (equal (INST-stg i) '(DQ 0))
                  (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (complete-stg-p (INST-stg (step-INST I MT MA sigs))))
     (complete-inst-robe-inv (step-INST i MT MA sigs)
                             (nth-robe (INST-tag (step-INST i MT MA sigs))
                                       (step-ROB MA sigs))))
  :hints (("goal" :in-theory (enable complete-inst-robe-inv lift-b-ops
                                     equal-b1p-converter
                                     INST-pc
                                     INST-sync?
                                     INST-WB-SREG?
                                     INST-wb?
                                     INST-rfeh?
                                     INST-BU?
                                     dispatch-inst?
                                     dispatch-cntlv
                                     step-robe)
                  :cases ((b1p (dispatch-inst? MA))))
          ("subgoal 1" :cases ((b1p (inst-fetch-error? i))))
          ("subgoal 1.2" :cases ((b1p (inst-decode-error? i))))))
)

; A help lemma to prove
;  not-complete-stg-step-inst-from-non-LSU-if-CDB-excpt-not-0
(defthm INST-data-accs-error-detected-p-step-INST-if-not-LSU
    (implies (and (inv MT MA)
                  (INST-p i) (INST-in i MT)
                  (or (IU-stg-p (INST-stg i))
                      (BU-stg-p (INST-stg i))
                      (MU-stg-p (INST-stg i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (not (INST-data-accs-error-detected-p
                   (step-inst i MT MA sigs))))
  :hints (("goal" :in-theory (e/d (INST-data-accs-error-detected-p
                                     INST-load-accs-error-detected-p
                                     INST-store-accs-error-detected-p
                                     equal-b1p-converter
                                     opcode-inst-type)
                                  (INST-IU-IF-IU-STG-P
                                   INST-BU-IF-BU-STG-P INST-MU-IF-MU-STG-P))
                  :use ((:instance INST-IU-IF-IU-STG-P)
```

```
                      (:instance INST-BU-IF-BU-STG-P)
                      (:instance INST-MU-IF-MU-STG-P)))))

; The instructions from IU, MU, and BU do not raise exceptions during
; the execution stage.  Thus, the excpt signal on the CDB is
; 0, when an instruction completes from one of them.
(defthm not-complete-stg-step-inst-from-non-LSU-if-CDB-excpt-not-0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (or (IU-stg-p (INST-stg i))
                      (MU-stg-p (INST-stg i))
                      (BU-stg-p (INST-stg i)))
                  (not (equal (CDB-excpt MA) 0)))
             (not (complete-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("goal" :in-theory (enable CDB-excpt step-INST
                                     lift-b-ops
                                     execute-stg-p
                                     issue-logic-def
                                     step-INST-execute
                                     step-INST-IU
                                     step-INST-BU
                                     step-INST-MU))))

; If an LSU instruction i completes this cycle,
; no new exception is detected in this cycle.
(defthm INST-data-accs-error-detected-p-step-INST-if-LSU-complete
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (LSU-stg-p (INST-stg i))
                  (complete-stg-p (INST-stg (step-INST i MT MA sigs)))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-data-accs-error-detected-p (step-INST i MT MA sigs))
                    (INST-data-accs-error-detected-p i)))
  :hints (("goal" :in-theory (e/d (INST-data-accs-error-detected-p
                                     INST-load-accs-error-detected-p
                                     INST-store-accs-error-detected-p
                                     LSU-field-lemmas
                                     equal-b1p-converter
                                     LSU-stg-p)
                                  (INST-load-OPCODE-3 INST-STORE-OPCODE-4
                                INST-load-OPCODE-6 INST-STORE-OPCODE-7))
                  :use ((:instance INST-STORE-OPCODE-7)
                        (:instance INST-STORE-OPCODE-4)
                        (:instance INST-load-OPCODE-3)
                        (:instance INST-load-OPCODE-6)))))

(encapsulate nil
(local
(defthm CDB-excpt-INST-excpt-flags-help
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (LSU-stg-p (INST-stg i))
                  (complete-stg-p (INST-stg (step-INST i MT MA sigs)))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (CDB-excpt MA)
                    (if (INST-data-accs-error-detected-p i) 6 0)))
  :hints (("goal" :in-theory (enable CDB-excpt INST-data-accs-error-detected-p
                                     INST-EXCPT-FLAGS LSU-stg-p
```

```
                                    CDB-FOR-LSU?)))))

; The exception field of the CDB contains the exception flags for the
; completing instruction.
(defthm CDB-excpt-INST-excpt-flags
    (implies (and (inv MT MA)
                  (execute-stg-p (INST-stg i))
                  (complete-stg-p (INST-stg (step-INST i MT MA sigs)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (equal (INST-excpt-flags (step-INST i MT MA sigs))
                    (CDB-excpt MA)))
    :hints (("goal" :in-theory (enable execute-stg-p INST-excpt-flags))))
)

; The LSB of the output-val signal from the BU is the branch outcome.
; See logcar-cdb-val-INST-branch-taken.
(defthm logcar-bu-output-val-inst-branch-taken
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (BU-stg-p (INST-stg i))
                  (or (b1p (BU-RS0-ISSUE-READY? (MA-BU MA)))
                      (b1p (BU-RS1-ISSUE-READY? (MA-BU MA))))
                  (equal (BU-output-tag (MA-BU MA)) (INST-tag i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (logcar (BU-output-val (MA-BU MA)))
                    (inst-branch-taken? i)))
  :hints (("goal" :in-theory (e/d (BU-output-val CDB-def
                                   BU-RS-field-lemmas
                                   INST-src-val3
                                   lift-b-ops
                                   INST-rc
                                   BU-stg-p bv-eqv-iff-equal
                                   equal-b1p-converter
                                   bu-output-def
                                   INST-BRANCH-TAKEN?
                                   issue-logic-def)
                                  ())
                  :use (:instance opcode-2-at-BU-stg-p))))

;  A output-dest signal from the IU outputs the Tomasulo's tag for the
;  completing integer instruction.
(defthm not-IU-output-tag-INST-tag-if-not-IU-stg-p
     (implies (and (inv MT MA)
                   (execute-stg-p (INST-stg i))
                   (not (IU-stg-p (INST-stg i)))
                   (or (b1p (IU-RS0-issue-ready? (MA-IU MA)))
                       (b1p (IU-RS1-issue-ready? (MA-IU MA))))
                   (INST-in i MT) (INST-p i)
                   (MAETT-p MT) (MA-state-p MA))
              (not (equal (IU-output-tag (MA-IU MA)) (INST-tag I))))
  :hints (("goal" :in-theory (enable IU-output-tag
                                   lift-b-ops
                                   IU-RS0-ISSUE-READY?
                                   IU-RS1-ISSUE-READY?
                                   IU-RS-field-lemmas))))

;  A output-dest signal from the BU outputs the Tomasulo's tag for the
```

```
;  completing branch instruction.
(defthm not-BU-output-tag-INST-tag-if-not-BU-stg-p
     (implies (and (inv MT MA)
                   (execute-stg-p (INST-stg i))
                   (not (BU-stg-p (INST-stg i)))
                   (or (b1p (BU-RS0-issue-ready? (MA-BU MA)))
                       (b1p (BU-RS1-issue-ready? (MA-BU MA))))
                   (INST-in i MT) (INST-p i)
                   (MAETT-p MT) (MA-state-p MA))
              (not (equal (BU-output-tag (MA-BU MA)) (INST-tag I))))
    :hints (("goal" :in-theory (enable BU-output-tag
                                       lift-b-ops
                                       BU-RS0-ISSUE-READY?
                                       BU-RS1-ISSUE-READY?
                                       BU-RS-field-lemmas))))

; The LSB of the val field of the common data bus is the branch
; outcome, if a branch instruction completes.
(defthm logcar-cdb-val-INST-branch-taken
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (b1p (INST-BU? i))
                  (b1p (CDB-ready-for? (INST-tag i) MA))
                  (execute-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (logcar (CDB-val MA)) (INST-branch-taken? i)))
  :hints (("goal" :in-theory (enable CDB-def lift-b-ops
                                     execute-stg-p
                                     LSU-field-lemmas
                                     BU-RS-field-lemmas
                                     IU-RS-field-lemmas
                                     MU-field-lemmas))))


; An important lemma for the proof of
;   complete-inst-robe-inv-step-INST
; The Instruction-ROB invariants are preserved for instruction i, if it
; advances from an execute-stg to a complete-stg.
;
; Some of the important lemmas for the proof for this lemmas are
; in section about Lemmas about CDB output.
(defthm complete-inst-robe-inv-step-INST-from-execute
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (execute-stg-p (INST-stg i))
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (complete-stg-p (INST-stg (step-INST I MT MA sigs))))
     (complete-inst-robe-inv (step-INST i MT MA sigs)
                             (nth-robe (INST-tag i)
                                       (step-ROB MA sigs))))
  :hints (("goal" :in-theory (enable complete-inst-robe-inv lift-b-ops
                                     INST-excpt-detected-p
                                     INST-PC
                                     INST-exunit-relations
                                     cdb-val-inst-dest-val
                                     equal-b1p-converter
```

```
                                    robe-receive-result? step-robe)
                  :cases ((b1p (CDB-ready-for? (INST-tag i) MA))))))

; If i is not at the head of the ROB, and some causes a jump,
; (MT-no-jmp-exintr-before i MT ...) is false.  Intuitively,
; there is an jump instruction that precedes i.
(defthm MT-no-jmp-exintr-before-complete-if-context-sync
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (complete-stg-p (INST-stg i))
                  (not (equal (inst-of-tag (MT-rob-head MT) MT) i))
                  (or (b1p (commit-jmp? MA))
                      (b1p (leave-excpt? MA))
                      (b1p (enter-excpt? MA)))
                  (MAETT-p MT) (MA-state-p MA) (INST-p i))
             (not (MT-no-jmp-exintr-before i MT MA sigs)))
  :hints (("goal" :use (:instance MT-jmp-exintr-before-if-inst-cause-jmp
                                  (j i)
                                  (i (inst-of-tag (MT-rob-head MT) MT))))))

; If flush-all? is on in the MA, and i is not a completed instruction, but
; it does not retire in this cycle, there must be another instruction that
; raises flush-all?.
(defthm MT-JMP-EXINTR-BEFORE-complete-INST-IF-FLUSH-ALL
    (implies (and (inv MT MA)
                  (INST-in i MT)
                  (complete-stg-p (INST-stg i))
                  (complete-stg-p (INST-stg (step-INST i MT MA sigs)))
                  (b1p (flush-all? MA sigs))
                  (MAETT-p MT) (MA-state-p MA) (INST-p i) (MA-input-p sigs))
             (not (MT-no-jmp-exintr-before i MT MA sigs)))
  :hints (("goal" :in-theory (enable flush-all? lift-b-ops)
                  :cases ((equal i (INST-OF-TAG (MT-ROB-HEAD MT) MT))))))

; Suppose i is a completed instruction at the head of the ROB. If commit-inst?
; is on in this cycle, i retires.
(defthm not-complete-stg-step-inst-if-INST-rob-not-head
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i)
                  (complete-stg-p (INST-stg i))
                  (b1p (commit-inst? MA))
                  (equal (INST-tag i) (MT-rob-head MT)))
             (not (complete-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("goal" :in-theory (enable step-INST step-INST-complete lift-b-ops
                                     INST-commit? bv-eqv-iff-equal))))

; The instruction-ROB invariants are preserved for instruction i,
; if i stays in a complete stage.
(defthm complete-inst-robe-inv-step-INST-from-complete
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (complete-stg-p (INST-stg i))
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (complete-stg-p (INST-stg (step-INST I MT MA sigs))))
     (complete-inst-robe-inv (step-INST i MT MA sigs)
                             (nth-robe (INST-tag i)
                                       (step-ROB MA sigs))))
  :hints (("goal" :in-theory (enable complete-inst-robe-inv
```

```
                                    INST-EXCPT-DETECTED-P
                                    step-robe lift-b-ops
                                    INST-pc)
                  :cases ((b1p (INST-fetch-error? i))))
          ("subgoal 2" :cases ((b1p (INST-decode-error? i))))
          ("subgoal 2.2" :cases ((b1p (INST-data-access-error? i))))))

; An important lemma.
; The instruction's ROB invariants are preserved for instruction i,
; if i will be in complete-stg in the next cycle.
;
; The proof needs three important lemmas
;    complete-inst-robe-inv-step-INST-from-DQ0
;    complete-inst-robe-inv-step-INST-from-execute
;    complete-inst-robe-inv-step-INST-from-complete
(defthm complete-inst-robe-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (complete-stg-p (INST-stg (step-INST I MT MA sigs))))
     (complete-inst-robe-inv (step-INST i MT MA sigs)
                             (nth-robe (INST-tag (step-INST i MT MA sigs))
                                       (step-ROB MA sigs))))
  :hints (("goal" :use (:instance stages-reachable-to-complete)
                  :in-theory (disable nth-robe-step-rob))))

; An important lemma for the case where i's next stage is (complete).
; (complete) is reachable from a bunch of execution stages.
; This lemma derives from complete-inst-robe-inv-step-INST.
(defthm complete-normal-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg (step-INST I MT MA sigs)) '(complete)))
            (complete-normal-inst-inv (step-INST i MT MA sigs)
                                      (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable complete-normal-inst-inv
                                     NOT-INST-STORE-IF-COMPLETE
                                     LSU-LOAD-IF-AT-LSU-RBUF-LCH)
                  :use ((:instance stages-reachable-to-complete-normal)
                        (:instance not-complete-stg-p-step-INST-if-INST-LSU)))))

; Invariants are preserved when i moves from (LSU wbuf0 lch) to
; (complete WBUF0).
(defthm complete-wbuf0-inst-inv-step-INST-LSU-wbuf0-lch
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg i) '(LSU wbuf0 lch))
                  (equal (INST-stg (step-INST I MT MA sigs))
```

```
                       '(complete WBUF0)))
            (complete-wbuf0-inst-inv (step-INST i MT MA sigs)
                                     (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable complete-wbuf0-inst-inv lift-b-ops
                                     LSU-STORE-IF-AT-LSU-WBUF
                                     update-wbuf0
                                     step-LSU step-wbuf0))))

; Invariants are preserved when i moves from (LSU wbuf1 lch) to
; (complete WBUF0).
(defthm complete-wbuf0-inst-inv-step-INST-LSU-wbuf1-lch
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg i) '(LSU wbuf1 lch))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(complete WBUF0)))
            (complete-wbuf0-inst-inv (step-INST i MT MA sigs)
                                     (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable complete-wbuf0-inst-inv lift-b-ops
                                     LSU-STORE-IF-AT-LSU-WBUF
                                     wbuf1-output update-wbuf0
                                     step-LSU step-wbuf0))))

; Invariants are preserved when i moves from (complete wbuf0) to
; (complete WBUF0).
(defthm complete-wbuf0-inst-inv-step-INST-complete-wbuf0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg i) '(complete wbuf0))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(complete WBUF0)))
            (complete-wbuf0-inst-inv (step-INST i MT MA sigs)
                                     (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable complete-wbuf0-inst-inv lift-b-ops
                                     LSU-STORE-IF-AT-LSU-WBUF
                                     step-LSU step-wbuf0
                                     update-wbuf0
                                     INST-COMMIT?))))

; Invariants are preserved when i moves from (complete wbuf1) to
; (complete WBUF0).
(defthm complete-wbuf0-inst-inv-step-INST-complete-wbuf1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg i) '(complete wbuf1))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(complete WBUF0)))
```

```
            (complete-wbuf0-inst-inv (step-INST i MT MA sigs)
                                     (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable complete-wbuf0-inst-inv lift-b-ops
                                     LSU-STORE-IF-AT-LSU-WBUF
                                     step-LSU step-wbuf0
                                     wbuf1-output
                                     INST-COMMIT?))))

; A important lemma for the case where i's next stage is (complete WBUF0).
; (complete WBUF0) is reachable from (complete wbuf1), (complete wbuf0),
; (LSU wbuf1 lch), and (LSU wbuf0 lch).
;
; The proof is divided into :
;   complete-wbuf0-inst-inv-step-INST-LSU-wbuf0-lch
;   complete-wbuf0-inst-inv-step-INST-LSU-wbuf1-lch
;   complete-wbuf0-inst-inv-step-INST-complete-wbuf0
;   complete-wbuf0-inst-inv-step-INST-complete-wbuf1
;
; It also uses complete-inst-robe-inv-step-INST.
(defthm complete-wbuf0-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(complete WBUF0)))
             (complete-wbuf0-inst-inv (step-INST i MT MA sigs)
                                      (MA-step MA sigs)))
  :hints (("goal" :use (:instance stages-reachable-to-complete-wbuf0))))

; The case where Instruction i goes from '(LSU WBUF1 lch)
; to '(complete WBUF1)
(defthm complete-wbuf1-inst-inv-step-INST-LSU-wbuf1-lch
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg i) '(LSU WBUF1 lch))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(complete WBUF1)))
             (complete-wbuf1-inst-inv (step-INST i MT MA sigs)
                                      (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable complete-wbuf1-inst-inv lift-b-ops
                                     LSU-STORE-IF-AT-LSU-WBUF
                                     step-LSU step-wbuf1
                                     wbuf1-output update-wbuf1
                                     INST-COMMIT?))))

; A case where Instruction goes from '(complete WBUF1) to '(complete WBUF1).
(defthm complete-wbuf1-inst-inv-step-INST-complete-wbuf1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
```

```
                (MT-no-jmp-exintr-before i MT MA sigs)
                (equal (INST-stg i) '(complete WBUF1))
                (equal (INST-stg (step-INST I MT MA sigs))
                       '(complete WBUF1)))
           (complete-wbuf1-inst-inv (step-INST i MT MA sigs)
                                    (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable complete-wbuf1-inst-inv lift-b-ops
                                     LSU-STORE-IF-AT-LSU-WBUF
                                     step-LSU step-wbuf1
                                     wbuf1-output update-wbuf1
                                     INST-COMMIT?))))

; A important lemma for the case where i's next stage is (complete WBUF1)
; The instruction invariants are preserved if the next stage is
; (complete WBUF1)
; The stage (complete WBUF1) is reachable from
; '(LSU WBUF1 lch) and '(complete WBUF1).
;
; The proof is divided into two parts:
;   complete-wbuf1-inst-inv-step-INST-LSU-wbuf1-lch
;   complete-wbuf1-inst-inv-step-INST-complete-wbuf1
; It also uses complete-inst-robe-inv-step-INST.
(defthm complete-wbuf1-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(complete WBUF1)))
             (complete-wbuf1-inst-inv (step-INST i MT MA sigs)
                                      (MA-step MA sigs)))
  :hints (("goal" :use (:instance stages-reachable-to-complete-wbuf1))))

; A landmark lemma complete-inst-inv-step-INST.
; The instruction invariants are preserved for instruction i, if
; i is in complete-stg in the next cycle.
(defthm complete-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (MT-no-jmp-exintr-before i MT MA sigs)
                  (NOT (B1P (INST-EXINTR-NOW? I MA SIGS)))
                  (complete-stg-p (INST-stg (step-INST I MT MA sigs))))
             (complete-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (e/d (complete-inst-inv complete-inst-inv
                                                     complete-stg-p)
                                  (nth-robe-step-rob)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Proof of commit-inst-inv
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; When an instruction at stage (complete wbuf0) commits, flush-all? is not
; asserted unless an exception is raised for the committed instruction.
; Note: This proof is slightly difficult.  One critical case is
; that the committed instruction is speculatively executed or self-modified.
; In that case, we can't prove the instruction is not a branch instruction,
```

```
; which may assert flush-all?.
(encapsulate nil
(local
(defthm not-flush-all-if-not-enter-excpt-help
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (or (equal (INST-stg i) '(complete wbuf0))
                      (equal (INST-stg i) '(complete wbuf1)))
                  (b1p (INST-commit? i MA))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (enter-excpt? MA)))
                  (MAETT-p MT) (MA-state-p MA))
             (not (b1p (flush-all? MA sigs))))
  :hints (("goal" :in-theory (enable flush-all? lift-b-ops
                                     commit-jmp? INST-commit?
                                     leave-excpt? ex-intr?)))))

(defthm not-flush-all-if-commit-INST-in
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (or (equal (INST-stg i) '(complete wbuf0))
                      (equal (INST-stg i) '(complete wbuf1)))
                  (b1p (INST-commit? i MA))
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (not (b1p (enter-excpt? MA)))
                  (MAETT-p MT) (MA-state-p MA))
             (not (b1p (flush-all? MA sigs))))
  :hints (("goal" :cases ((b1p (inst-specultv? i)) (b1p (INST-modified? i)))))
  :rule-classes
  ((:rewrite :corollary
     (implies (and (inv MT MA)
                   (INST-in i MT) (INST-p i)
                   (or (equal (INST-stg i) '(complete wbuf0))
                       (equal (INST-stg i) '(complete wbuf1)))
                   (b1p (flush-all? MA sigs))
                   (not (MT-CMI-p (MT-step MT MA sigs)))
                   (not (b1p (enter-excpt? MA)))
                   (MAETT-p MT) (MA-state-p MA))
              (equal (INST-commit? i MA) 0))
     :hints (("goal" :in-theory (enable equal-b1p-converter))))))
)

(in-theory (disable not-flush-all-if-commit-INST-in))

(defthm commit-wbuf0-inst-inv-step-INST-complete-wbuf0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (equal (INST-stg i) '(complete wbuf0))
                  (equal (INST-stg (step-INST I MT MA sigs)) '(commit wbuf0)))
             (commit-wbuf0-inst-inv (step-INST i MT MA sigs)
                                    (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable commit-wbuf0-inst-inv lift-b-ops
                                     step-LSU step-wbuf0
                                     not-flush-all-if-commit-INST-in
                                     LSU-STORE-IF-AT-LSU-WBUF
                                     update-wbuf0)
```

```
                  :cases ((b1p (inst-specultv? i))
                          (b1p (INST-modified? i))))
          (when-found (EQUAL (INST-tag I) (MT-ROB-HEAD MT))
                      (:in-theory (enable INST-commit? lift-b-ops)))
          (when-found (b1p (commit-inst? MA))
                      (:in-theory (enable INST-commit? lift-b-ops)))))

(defthm commit-wbuf0-inst-inv-step-INST-complete-wbuf1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (equal (INST-stg i) '(complete wbuf1))
                  (equal (INST-stg (step-INST I MT MA sigs)) '(commit wbuf0)))
             (commit-wbuf0-inst-inv (step-INST i MT MA sigs)
                                    (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable commit-wbuf0-inst-inv lift-b-ops
                                     step-LSU step-wbuf0
                                     not-flush-all-if-commit-INST-in
                                     LSU-STORE-IF-AT-LSU-WBUF
                                     wbuf1-output)
                  :cases ((b1p (inst-specultv? i))
                          (b1p (INST-modified? i))))
          (when-found (EQUAL (INST-tag I) (MT-ROB-HEAD MT))
                      (:in-theory (enable INST-commit? lift-b-ops)))
          (when-found (b1p (commit-inst? MA))
                      (:in-theory (enable INST-commit? lift-b-ops)))))

(defthm commit-wbuf0-inst-inv-step-INST-commit-wbuf0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (equal (INST-stg i) '(commit wbuf0))
                  (equal (INST-stg (step-INST I MT MA sigs)) '(commit wbuf0)))
             (commit-wbuf0-inst-inv (step-INST i MT MA sigs)
                                    (MA-step MA sigs)))
    :hints (("goal" :in-theory (enable commit-wbuf0-inst-inv lift-b-ops
                                       not-flush-all-if-commit-INST-in
                                       step-LSU step-wbuf0
                                       update-wbuf0
                                       LSU-STORE-IF-AT-LSU-WBUF)
                    :cases ((b1p (inst-specultv? i))
                            (b1p (INST-modified? i))))
          (when-found (EQUAL (INST-tag I) (MT-ROB-HEAD MT))
                      (:in-theory (enable INST-commit? lift-b-ops)))
          (when-found (b1p (commit-inst? MA))
                      (:in-theory (enable INST-commit? lift-b-ops)))))

(defthm commit-wbuf0-inst-inv-step-INST-commit-wbuf1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (equal (INST-stg i) '(commit wbuf1))
```

```
                (equal (INST-stg (step-INST I MT MA sigs)) '(commit wbuf0)))
           (commit-wbuf0-inst-inv (step-INST i MT MA sigs)
                                  (MA-step MA sigs)))
     :hints (("goal" :in-theory (enable commit-wbuf0-inst-inv lift-b-ops
                                        not-flush-all-if-commit-INST-in
                                        step-LSU step-wbuf0
                                        wbuf1-output
                                        LSU-STORE-IF-AT-LSU-WBUF))
          (when-found (EQUAL (INST-tag I) (MT-ROB-HEAD MT))
                      (:in-theory (enable INST-commit? lift-b-ops)))
          (when-found (b1p (commit-inst? MA))
                      (:in-theory (enable INST-commit? lift-b-ops)))))

(defthm commit-wbuf0-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (equal (INST-stg (step-INST I MT MA sigs)) '(commit wbuf0)))
           (commit-wbuf0-inst-inv (step-INST i MT MA sigs)
                                  (MA-step MA sigs)))
  :hints (("goal" :use (:instance stages-reachable-to-commit-wbuf0)
                  :in-theory (enable lift-b-ops))))

(defthm commit-wbuf1-inst-inv-step-INST-complete-wbuf1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (equal (INST-stg i) '(complete wbuf1))
                  (equal (INST-stg (step-INST I MT MA sigs)) '(commit wbuf1)))
           (commit-wbuf1-inst-inv (step-INST i MT MA sigs)
                                  (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable commit-wbuf1-inst-inv lift-b-ops
                                     not-flush-all-if-commit-INST-in
                                     step-LSU step-wbuf1
                                     update-wbuf1
                                     LSU-STORE-IF-AT-LSU-WBUF)
                  :cases ((b1p (inst-specultv? i))
                          (b1p (INST-modified? i))))
          (when-found (EQUAL (INST-tag I) (MT-ROB-HEAD MT))
              (:in-theory (enable INST-commit? lift-b-ops)))
          (when-found (b1p (commit-inst? MA))
                      (:in-theory (enable INST-commit? lift-b-ops)))))

(defthm commit-wbuf1-inst-inv-step-INST-commit-wbuf1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (equal (INST-stg i) '(commit wbuf1))
                  (equal (INST-stg (step-INST I MT MA sigs)) '(commit wbuf1)))
           (commit-wbuf1-inst-inv (step-INST i MT MA sigs)
```

```
                                    (MA-step MA sigs)))
    :hints (("goal" :in-theory (enable commit-wbuf1-inst-inv lift-b-ops
                                       not-flush-all-if-commit-INST-in
                                       step-LSU step-wbuf1
                                       update-wbuf1
                                       LSU-STORE-IF-AT-LSU-WBUF))
            (when-found (EQUAL (INST-tag I) (MT-ROB-HEAD MT))
                        (:in-theory (enable INST-commit? lift-b-ops)))
            (when-found (b1p (commit-inst? MA))
                        (:in-theory (enable INST-commit? lift-b-ops)))))

(defthm commit-wbuf1-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (equal (INST-stg (step-INST I MT MA sigs)) '(commit wbuf1)))
             (commit-wbuf1-inst-inv (step-INST i MT MA sigs)
                                    (MA-step MA sigs)))
  :hints (("goal" :use (:instance stages-reachable-to-commit-wbuf1)
                  :in-theory (enable lift-b-ops))))

; A landmark lemma commit-inst-inv-step-INST
(defthm commit-inst-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (INST-inv i MA)
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (commit-stg-p (INST-stg (step-INST I MT MA sigs))))
             (commit-inst-inv (step-INST i MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable commit-inst-inv commit-stg-p))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   End of the proof of commit-inst-inv
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; The final landmark lemma.
; The instruction invariants are preserved for instruction i, regardless
; of the stage of i, if external interrupt does not interrupt i.
(defthm INST-inv-step-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (INST-p i)
                  (INST-in i MT)
                  (not (b1p (INST-exintr-now? i MA sigs)))
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (MT-no-jmp-exintr-before i MT MA sigs))
             (INST-inv (step-INST i MT MA sigs)
                       (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable INST-inv))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  End of the proof of INST-inv-step-INST
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Proof of INST-inv-exintr-INST
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; A landmark lemma.
; The instruction invariants are true for (exintr-INST MT init smc).
(defthm INST-inv-exintr-INST
    (implies (and (MAETT-p MT)
                  (MA-state-p MA)
                  (ISA-state-p INIT)
                  (MA-input-p sigs))
             (INST-inv (exintr-INST MT init smc) (MA-step MA sigs)))
  :hints (("Goal" :in-theory (enable exintr-INST inst-inv-def))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; End of inst-inv
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Proof of MT-INST-inv
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Induction on the list of instructions.
(defthm trace-INST-inv-lemma
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-listp trace)
                  (MA-input-p sigs)
                  (MT-no-jmp-exintr-before-trace trace MT MA sigs)
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (subtrace-p trace MT)
                  (trace-INST-inv trace MA))
     (trace-INST-inv (step-trace trace MT MA sigs
                                 (ISA-before trace MT)
                                 (specultv-before? trace MT)
                                 (modified-inst-before? trace MT))
                     (MA-step MA sigs)))
  :hints ((when-found (FETCHED-INST MT (MT-FINAL-ISA MT)
                                    (MT-SPECULTV? MT))
                      (:cases ((b1p (MT-SPECULTV? MT)))))))

(encapsulate nil
(local
(defthm MT-INST-inv-preserved-help
    (implies (and (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (inv MT MA)
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (MT-INST-inv MT MA))
             (MT-INST-inv (MT-step MT ma sigs) (MA-step ma sigs)))
  :hints (("goal" :use (:instance trace-INST-inv-lemma
                                  (trace (MT-trace MT)))
                  :in-theory (enable MT-INST-inv
                                     MT-step)))))
; MT-INST-inv is preserved.
(defthm MT-INST-inv-preserved
    (implies (and (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (inv MT MA))
             (MT-INST-inv (MT-step MT ma sigs)
                          (MA-step ma sigs))))
) ; encapsulate
```

D.6.6 reg-ref.lisp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MI-inv.lisp
; Author  Jun Sawada, University of Texas at Austin
;
;  This book proves the invariant properties consistent-reg-tbl-p and
;  consistent-sreg-tbl-p, and consistent-RS-entry-p.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(in-package "ACL2")

(include-book "MA2-lemmas")
(include-book "MAETT-lemmas")
(include-book "modifier")

(deflabel begin-reg-ref)
;  This book proves the invariant condition about Tomasulo's tags.
;
;  Index.
;    Proof of consistent-reg-tbl-p
;      Base case
;        consistent-reg-ref-p-init-MT
;      Induction Case
;          not-LRM-in-ROB-MT-step-if-not-reg-ref-wait
;          LRM-in-ROB-MT-step-if-reg-ref-wait
;          LRM-in-ROB-MT-step
;            INST-tag-LRM-MT-step-if-not-dispatch-inst
;            INST-tag-LRM-MT-step-if-not-reg-modifier
;            INST-tag-LRM-MT-step-if-dispatch-inst
;          consistent-reg-ref-p-MT-step
;        consistent-reg-tbl-p-preserved
;    Proof of consistent-sreg-tbl-p
;      Base Case
;        consistent-sreg-tbl-p-init-MT
;      Inductive case (not yet commented properly)
;           not-LSRM-in-ROB-MT-step-if-not-reg-ref-wait
;           LSRM-in-ROB-MT-step-if-reg-ref-wait
;           LSRM-in-ROB-MT-step
;             INST-tag-LSRM-MT-step-if-not-dispatch-inst
;             INST-tag-LSRM-MT-step-if-not-sreg-modifier
;             INST-tag-LSRM-MT-step-if-dispatch-inst
;           INST-tag-LSRM-MT-step
;         consistent-sreg-ref-p-MT-step
;        consistent-sreg-tbl-p-preserved
;
;    Proof of consistent-RS-p
;        Rules about Common Data Bus and instruction stage.
;        Rewriting Rules derived from consistent-RS-p
;        Proof of the existence of the register modifiers.
;          e.g. exist-LRM-before-p-step-INST-operand0-ra
;        Proof of lemmas about dispatched instructions.
;          e.g. INST-tag-LRM-before-step-INST-logbit0-ra
;        Proof of lemmas about idling instructions.
;          e.g. execute-stg-p-LRM-before-step-INST-if-logbit0-ra
;        Case analysis on the old and new stage of i
;          e.g. consistent-IU-RS0-p-step-INST-DQ0
;      consistent-RS-entry-p-step-INST
;

(in-theory (disable (:REWRITE NOT-ROB-EMPTY-IF-INST-IS-EXECUTED . 1)
                    NOT-EX-INTR-IF-ROB-NOT-EMPTY))
```

```
(in-theory
 (disable (:rewrite not-member-equal-cdr-if-car-is-not-dispatched . 1)
          (:rewrite not-member-equal-cdr-if-car-is-not-commit . 1)
          (:rewrite INST-at-DQ-0-is-first-non-dispatched-inst . 1)))

(in-theory
 (enable (:rewrite not-member-equal-cdr-if-car-is-not-dispatched . 2)
         (:rewrite not-member-equal-cdr-if-car-is-not-commit . 2)
         (:rewrite INST-at-DQ-0-is-first-non-dispatched-inst . 2)))

(local
(defthm uniq-inst-at-DQ0-if-dispatch-inst
    (implies (and (inv MT MA)
                  (b1p (dispatch-inst? MA))
                  (MAETT-p MT) (MA-state-p MA))
             (uniq-inst-at-stg '(DQ 0) MT))
  :hints (("Goal" :in-theory (enable lift-b-ops
                                     DQ-ready-no-unit? DQ-ready-to-IU?
                                     DQ-ready-to-MU? DQ-ready-to-BU?
                                     DQ-ready-to-LSU?
                                     dispatch-inst? dispatch-no-unit?
                                     dispatch-to-IU? dispatch-to-MU?
                                     dispatch-to-LSU? dispatch-to-BU?)))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Proof of consistent-reg-tbl-p
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Proof of consistent-reg-tbl-p for initial states
(encapsulate nil
(local
(defthm reg-ref-empty-if-MA-flushed-help
    (implies (and (b1p (reg-tbl-empty-under? index2 DQ))
                  (integerp index1) (integerp index2)
                  (<= 0 index1) (< index1 index2))
             (b1p (reg-ref-empty? index1 DQ)))
  :hints (("goal" :in-theory (enable lift-b-ops)))))

; If MA is flushed, the valid flags of register reference table are all 0.
(defthm reg-ref-empty-if-MA-flushed
    (implies (and (b1p (MA-flushed? MA)) (rname-p index))
             (b1p (reg-ref-empty? index (MA-DQ MA))))
  :hints (("goal" :in-theory (enable MA-flushed? DQ-empty?
                                     lift-b-ops rname-p
                                     reg-tbl-empty?))))
)

(encapsulate nil
(local
(defthm consistent-reg-ref-p-init-MT-help
    (implies (and (MA-state-p MA) (b1p (reg-ref-empty? index (MA-DQ MA))))
             (consistent-reg-ref-p index (init-MT MA) MA))
  :hints (("goal" :in-theory (enable reg-ref-empty? lift-b-ops
                                     consistent-reg-ref-p
                                     INIT-MT exist-LRM-in-ROB-p)))))

; Register reference table satisfies consistent-reg-ref-p,
; when MA is flushed.  Simply a clean (no valid entry) reference table is
; consistent.
(defthm consistent-reg-ref-p-init-MT
    (implies (and (MA-state-p MA) (b1p (MA-flushed? MA))  (rname-p index))
             (consistent-reg-ref-p index (init-MT MA) MA)))
)
```

```
(encapsulate nil
(local
(defthm consistent-reg-tbl-p-init-MT-help
    (implies (and (MA-state-p MA) (b1p (MA-flushed? MA))
                  (integerp index) (<= 0 index) (<= index *num-regs*))
             (consistent-reg-tbl-under index (init-MT MA) MA))
  :hints (("goal" :in-theory (enable rname-p unsigned-byte-p)))))

; consistent-reg-tbl-p is true for the initial MAETT.
(defthm consistent-reg-tbl-p-init-MT
    (implies (and (MA-state-p MA) (b1p (MA-flushed? MA)))
             (consistent-reg-tbl-p (init-MT MA) MA))
  :hints (("goal" :in-theory (enable consistent-reg-tbl-p))))
)

;; Invariant proof
(encapsulate nil
(local
(defun induct-help (i tbl)
  (if (endp tbl)
      (cons i tbl)
      (induct-help (1+ i) (cdr tbl)))))

(local
(defthm rname-idempotent
    (implies (and (integerp i) (<= 0 i)  (< i *num-regs*))
             (equal (rname i) i))
  :hints (("Goal" :in-theory (enable rname loghead)))))

(local
(defthm reg-tbl-nth-step-DQ-help
    (implies (and (integerp i) (<= 0 i)
                  (integerp idx) (<= 0 idx)
                  (<= i idx) (< idx *num-regs*)
                  (< (- idx i) (len tbl)))
             (equal (nth (- idx i) (step-reg-list tbl i MA sigs))
                    (step-reg-ref (nth (- idx i) tbl)
                                  idx MA sigs)))
  :hints (("goal" :induct (induct-help i tbl)
                  :in-theory (e/d (STEP-REG-LIST) ()))
          (when-found (car tbl)
                      (:expand (STEP-REG-LIST TBL I MA SIGS)))
          (when-found (cdr tbl)
                      (:expand (STEP-REG-LIST TBL I MA SIGS))))
  :rule-classes nil))

; This is a rule to open up the definition of step-DQ.
(defthm reg-tbl-nth-step-DQ
    (implies (and (MA-state-p MA) (rname-p idx))
             (equal (reg-tbl-nth idx (DQ-reg-tbl (step-DQ MA sigs)))
                    (step-reg-ref (reg-tbl-nth idx (DQ-reg-tbl (MA-DQ MA)))
                                  idx MA sigs)))
  :hints (("goal" :in-theory (enable step-DQ step-reg-tbl unsigned-byte-p
                                     rname-p reg-tbl-nth)
                  :use (:instance reg-tbl-nth-step-DQ-help
                                  (idx idx) (i 0)
                                  (tbl (DQ-reg-tbl (MA-DQ MA)))))))
)

; The instruction i is the oldest non-committed instruction.  Only when i
; commits, flush-all? can be 1.  I must be the instruction that causes the
; flush-all.
```

```
(defthm commit-inst-step-inst-if-flush-all
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MT-all-commit-before i MT)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (INST-exintr-now? i MA sigs)))
                  (b1p (flush-all? MA sigs)))
             (committed-p (step-INST i MT MA sigs)))
  :hints (("Goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (COMMITTED-P)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm not-LRM-in-ROB-MT-step-if-flush-all-help
    (implies (and (inv MT MA)
                  (MT-all-commit-before-trace trace MT)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (flush-all? MA sigs)))
             (not (trace-exist-LRM-in-ROB-p idx
                                            (step-trace trace MT MA sigs
                                                        ISA spc smc))))
  :hints (("goal" :in-theory (enable committed-p))
          (when-found (MT-ALL-COMMIT-BEFORE-TRACE (CDR TRACE) MT)
                      (:cases ((committed-p (car trace))))))))

; No register modifier will be in the ROB after a flush-all.
; This is used to prove the consistent-reg-ref-p
(defthm not-LRM-in-ROB-MT-step-if-flush-all
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (flush-all? MA sigs)))
             (not (exist-LRM-in-ROB-p idx (MT-step MT MA sigs))))
  :hints (("Goal" :in-theory (enable exist-LRM-in-ROB-p))))
)

; To determine whether the newly dispatched instruction is a register
; modifier, we only have to check the wb? field and wb-sreg? field of
; the control vector coming out of dispatcher. (dispatch-cntlv MA)
; is the cntlv vector output from the dispatch logic.  If wb? is 0
; or wb-sreg is 1, the dispatched instruction is not a register
; modifier.
(defthm not-reg-modifier-if-not-cntlv-wb
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (not (b1p (INST-specultv? (INST-at-stg '(DQ 0) MT))))
                  (not (b1p (INST-modified? (INST-at-stg '(DQ 0) MT))))
                  (not (INST-fetch-error-detected-p (INST-at-stg '(DQ 0) MT)))
                  (or (not (b1p (cntlv-wb? (dispatch-cntlv MA))))
                      (b1p (cntlv-wb-sreg? (dispatch-cntlv MA)))))
             (not (reg-modifier-p idx (INST-at-stg '(DQ 0) MT))))
  :hints (("Goal" :in-theory (enable dispatch-cntlv reg-modifier-p
                                     dispatch-inst? DISPATCH-NO-UNIT?
                                     dispatch-to-IU? dispatch-to-MU?
                                     dispatch-to-BU? dispatch-to-LSU?
                                     DQ-READY-NO-UNIT? DQ-READY-TO-BU?
                                     DQ-READY-TO-IU? DQ-READY-TO-LSU?
                                     DQ-READY-TO-MU?
                                     INST-WB-SREG? INST-WB?
                                     lift-b-ops))))
```

```
(local
(defthm not-trace-LRM-in-ROB-MT-step-if-no-dispatch
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (rname-p idx)
                  (not (trace-exist-LRM-in-ROB-p idx trace))
                  (not (b1p (dispatch-inst? MA))))
             (not (trace-exist-LRM-in-ROB-p idx
                       (step-trace trace MT MA sigs ISA spc smc))))))

; There will be no register modifier in the ROB in the next cycle
; if no modifiers are in the ROB in this cycle, and no new instruction is
; dispatched.
(defthm not-LRM-in-ROB-MT-step-if-no-dispatch
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (rname-p idx)
                  (not (exist-LRM-in-ROB-p idx MT))
                  (not (b1p (dispatch-inst? MA))))
             (not (exist-LRM-in-ROB-p idx (MT-step MT MA sigs))))
  :hints (("Goal" :in-theory (enable exist-LRM-in-ROB-p MT-step))))

; If i is not at DQ0, it won't be dispatched in this cycle.
(defthm not-dispatched-inst-step-inst-if-not-DQ0
    (implies (and (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (INST-p i)
                  (not (dispatched-p i))
                  (not (equal (INST-stg i) '(DQ 0))))
             (not (dispatched-p (step-inst i MT MA sigs))))
  :hints (("Goal" :in-theory (enable dispatched-p*
                                     step-inst-dq-inst
                                     step-inst-dq
                                     DQ-STG-P))))

(in-theory (enable uniq-inst-at-INST-stg-if-INST-in))

(local
(encapsulate nil
(local
 (defthm local-help
     (implies (and (inv MT MA)
                   (INST-in i MT) (INST-p i)
                   (INST-in j MT) (INST-p j)
                   (INST-in-order-p i j MT)
                   (not (equal i j))
                   (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                   (DQ-stg-p (INST-stg i)) (DQ-stg-p (INST-stg j)))
              (DQ-stg-p (INST-stg (step-INST j MT MA sigs))))
 :hints (("Goal" :in-theory (enable DQ-stg-p step-inst-dq-inst
                                    step-inst-low-level-functions)
                 :use ((:instance uniq-stage-inst)
                       (:instance DQ-stg-index-monotonic))))))

(local
 (defthm not-dispatched-inst-step-inst-if-earlier-non-dispatched
     (implies (and (inv MT MA)
                   (INST-in i MT) (INST-p i)
                   (INST-in j MT) (INST-p j)
                   (INST-in-order-p i j MT)
                   (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
```

```
                (not (dispatched-p i)))
           (not (dispatched-p (step-inst j MT MA sigs))))
   :hints (("Goal" :in-theory (e/d (dispatched-p* IFU-IS-LAST-INST)
                                   (inst-is-at-one-of-the-stages))
                   :restrict ((local-help ((i i))))
                   :use (:instance inst-is-at-one-of-the-stages
                         (i j)))
           (when-found (DQ-stg-p (INST-stg j))
                       (:cases ((equal i j)))))))

; A local lemma.
(defthm not-trace-LRM-in-ROB-step-trace
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (INST-in i MT) (INST-p i)
                  (not (dispatched-p i))
                  (subtrace-after-p i trace MT)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (not (trace-exist-LRM-in-ROB-p idx
                        (step-trace trace MT MA sigs ISA spc smc)))))
))

(local
(defthm not-trace-LRM-in-ROB-MT-step-if-not-reg-modifier
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (rname-p idx)
                  (not (trace-exist-LRM-in-ROB-p idx trace))
                  (not (reg-modifier-p idx (INST-at-stg-in-trace '(DQ 0)
                                                                 trace))))
             (not (trace-exist-LRM-in-ROB-p idx
                         (step-trace trace MT MA sigs ISA spc smc))))
  :hints (("Goal" :in-theory (enable dispatched-p)
                  :restrict ((not-trace-LRM-in-ROB-step-trace
                              ((i (car trace)))))))))

; If there is no register modifier in the ROB in the current cycle, and the
; instruction at DQ0 is not a register modifier,
; there will be no register modifier in the ROB in the next cycle.
(defthm not-LRM-in-ROB-MT-step-if-not-reg-modifier
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (rname-p idx)
                  (not (exist-LRM-in-ROB-p idx MT))
                  (not (reg-modifier-p idx (INST-at-stg '(DQ 0) MT))))
             (not (exist-LRM-in-ROB-p idx (MT-step MT MA sigs))))
  :hints (("Goal" :in-theory (enable exist-LRM-in-ROB-p
                                     INST-at-stg))))

; INST-start-specultv? for a particular instruction i does not
; change after stepping the instruction.
;
; INST-start-specultv? is 1 for the instruction that starts
; speculative execution. For instance, a mispredicted branch starts a
; speculation.  Thus, instruction in IFU stage may change the value of
; INST-start-specultv? when branch prediction makes a wrong choice.
; Another possibility is that the instruction abandons the following
; instructions and starts the execution of the correct branch all over
; again.  This happens only when an instruction commits, and it happens
; only when flush-all? is 1. Except these cases, the value
; of INST-start-specultv?  does not change.
```

```
;
; Note:
; If the instruction is at the execute stage, this is true from the
; definition of INST-start-specultv?.  If it is at complete-stg-p,
; the instruction will not retire because flush-all? is off.  Therefore,
; the value of INST-start-specultv? does not change.
(defthm INST-start-speculative-if-not-flush-all
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (dispatched-p i)
                  (not (committed-p i))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (INST-specultv? i)))
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (equal (INST-start-specultv? (step-INST i MT MA sigs))
                    (INST-start-specultv? i)))
  :hints (("Goal" :in-theory (e/d (dispatched-p committed-p
                                   equal-b1p-converter)
                                  (inst-is-at-one-of-the-stages))
                  :cases ((b1p (INST-modified? i))))
          ("subgoal 1" :cases ((committed-p (step-INST i MT MA sigs))))
          ("subgoal 1.1" :in-theory (e/d (dispatched-p committed-p
                                          step-inst-low-level-functions
                                          step-inst-complete-inst
                                          lift-b-ops
                                          equal-b1p-converter
                                   step-inst-complete-inst)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm MT-modified-at-dispatch-MT-step-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (b1p (trace-modified-at-dispatch? trace))
                  (not (b1p (flush-all? MA sigs)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (equal (trace-modified-at-dispatch?
                      (step-trace trace MT MA sigs ISA spc smc))
                    1))))

; If the processor is executing modified instruction stream at
; the dispatching boundary, it will continue to execute modified stream
; unless flush-all? is true.
(defthm MT-modified-at-dispatch-MT-step
    (implies (and (inv MT MA)
                  (b1p (MT-modified-at-dispatch? MT))
                  (not (b1p (flush-all? MA sigs)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (equal (MT-modified-at-dispatch? (MT-step MT MA sigs)) 1))
  :hints (("goal" :in-theory (enable MT-modified-at-dispatch?))))
)

; If the processor is executing speculatively at the dispatching boundary.
; the instruction at (DQ 0) is a speculative instruction.
(defthm INST-specultv-INST-at-DQ0-if-MT-specultv-at-dispatch
    (implies (and (inv MT MA)
                  (b1p (MT-specultv-at-dispatch? MT))
                  (uniq-inst-at-stg '(DQ 0) MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-specultv? (INST-at-stg '(DQ 0) MT)) 1))
```

```
  :hints (("Goal"
           :restrict
           ((MT-specultv-at-dispatch-off-if-non-specultv-inst-in
             ((i (INST-at-stg '(DQ 0) MT)))))
           :in-theory
           (enable dispatched-p
                   MT-specultv-at-dispatch-off-if-non-specultv-inst-in
                   equal-b1p-converter))))

; The processor is executing modified instruction stream at the
; dispatch boundary, the instruction at DQ0 is in the modified instruction
; stream.
(defthm INST-modified-INST-at-DQ0-if-MT-CMI
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg '(DQ 0) MT)
                  (b1p (MT-modified-at-dispatch? MT)))
             (equal (INST-modified? (INST-at-stg '(DQ 0) MT)) 1))
  :hints (("Goal" :in-theory
                  (enable equal-b1p-converter
                  INST-modified-at-dispatch-off-if-undispatched-inst-in)
                  :restrict
                  ((INST-modified-at-dispatch-off-if-undispatched-inst-in
                    ((i (INST-at-stg '(DQ 0) MT))))))))

(encapsulate nil
(local
(defthm DQ0-is-dispatched
    (implies (and (INST-p i) (equal (INST-stg i) '(DQ 0)))
             (not (dispatched-p i)))
  :hints (("Goal" :in-theory (enable dispatched-p)))))

(local
(defthm MT-specultv-at-dispatch-MT-step-if-specultv-dispatch-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (member-equal i trace) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (dispatch-inst? MA))
                  (b1p (INST-specultv? i))
                  (equal (INST-stg i) '(DQ 0))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (trace-specultv-at-dispatch?
                      (step-trace trace MT MA sigs ISA spc smc))
                    1))
  :hints (("Goal" :in-theory (enable lift-b-ops)))))

; If the instruction at DQ0 is speculatively executed, and the
; instruction is dispatched in this cycle, the processor will be
; executing a speculative instruction stream at the dispatching
; boundary.
(defthm MT-specultv-at-dispatch-MT-step-if-specultv-dispatch
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (INST-specultv? i))
                  (b1p (dispatch-inst? MA))
                  (equal (INST-stg i) '(DQ 0))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (MT-specultv-at-dispatch? (MT-step MT MA sigs)) 1))
  :hints (("Goal" :in-theory (e/d (MT-specultv-at-dispatch?
                                   INST-in)
```

```
                                  ()))))
)

; If the processor is not executing speculatively at the dispatching
; boundary, the instruction at DQ0 is not speculatively executed.
(defthm INST-specultv-INST-at-DQ0-if-dispatch-inst
    (implies (and (inv MT MA)
                  (b1p (dispatch-inst? MA))
                  (not (b1p (flush-all? MA sigs)))
                  (uniq-inst-at-stg '(DQ 0) MT)
                  (not (b1p (MT-specultv-at-dispatch?
                             (MT-step MT MA sigs))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-specultv? (INST-at-stg '(DQ 0) MT)) 0))
  :hints (("Goal" :in-theory (enable equal-b1p-converter)
                  :restrict
                  ((MT-specultv-at-dispatch-MT-step-if-specultv-dispatch
                    ((i (INST-at-stg '(DQ 0) MT))))))))

(encapsulate nil
(local
(defthm MT-specultv-at-dispatch-MT-step-if-fetch-error-detected-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (equal (INST-stg i) '(DQ 0))
                  (member-equal i trace) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (dispatch-inst? MA))
                  (b1p (INST-start-specultv? i))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (equal (trace-specultv-at-dispatch?
                      (step-trace trace MT MA sigs ISA spc smc))
                    1))
  :hints (("Goal" :in-theory (enable lift-b-ops)))))

; If a fetch error has been detected with the dispatched instruction,
; then the processor starts executing instructions speculatively at the
; dispatching boundary from the next cycle.   Note that the
; instructions following an exception are considered to be speculatively
; executed.
(defthm MT-specultv-at-dispatch-MT-step-if-fetch-error-detected
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (dispatch-inst? MA))
                  (INST-fetch-error-detected-p i)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (equal (MT-specultv-at-dispatch? (MT-step MT MA sigs))
                    1))
  :hints (("Goal" :in-theory (enable INST-start-specultv? INST-excpt?
                                     INST-in lift-b-ops
                                     MT-specultv-at-dispatch?)
                  :cases ((b1p (INST-fetch-error? i))))))
)

; Contrapositive of MT-specultv-at-dispatch-MT-step-if-fetch-error-detected
(defthm INST-fetch-error-INST-at-DQ0-if-dispatch-inst
    (implies (and (inv MT MA)
                  (b1p (dispatch-inst? MA))
                  (not (b1p (flush-all? MA sigs)))
                  (uniq-inst-at-stg '(DQ 0) MT)
```

```
                (not (b1p (MT-specultv-at-dispatch?
                           (MT-step MT MA sigs))))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (not (INST-fetch-error-detected-p (INST-at-stg '(DQ 0) MT))))
  :hints (("Goal" :restrict
                ((MT-specultv-at-dispatch-MT-step-if-fetch-error-detected
                  ((i (INST-at-stg '(DQ 0) MT))))))))

(encapsulate nil
(local
(defthm MT-modified-at-dispatch-MT-step-if-dispatch-inst-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (member-equal i trace) (INST-p i)
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                (not (b1p (flush-all? MA sigs)))
                (b1p (dispatch-inst? MA))
                (equal (INST-stg i) '(DQ 0))
                (b1p (INST-modified? i))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (equal (trace-modified-at-dispatch?
                   (step-trace trace MT MA sigs ISA spc smc))
                  1))))

; If instruction at DQ0 is a modified instruction that is dispatched in
; this cycle, the processor will be executing a modified stream of
; instructions at dispatching boundary in the next cycle.
(defthm MT-modified-at-dispatch-MT-step-if-dispatch-inst
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                (not (b1p (flush-all? MA sigs)))
                (b1p (dispatch-inst? MA))
                (equal (INST-stg i) '(DQ 0))
                (b1p (INST-modified? i)))
           (equal  (MT-modified-at-dispatch? (MT-step MT MA sigs)) 1))
  :hints (("Goal" :in-theory (enable MT-modified-at-dispatch? INST-in))))
)

; Contrapositive of MT-modified-at-dispatch-MT-step-if-dispatch-inst
(defthm INST-modified-INST-at-DQ0-if-dispatch-inst
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                (not (b1p (flush-all? MA sigs)))
                (uniq-inst-at-stg '(DQ 0) MT)
                (b1p (dispatch-inst? MA))
                (not (b1p (MT-modified-at-dispatch? (MT-step MT MA sigs)))))
           (equal (INST-modified? (INST-at-stg '(DQ 0) MT)) 0))
  :hints (("Goal" :restrict
                ((MT-modified-at-dispatch-MT-step-if-dispatch-inst
                  ((i (INST-at-stg '(DQ 0) MT)))))
                :in-theory (enable equal-b1p-converter))))

(local
(defthm MT-specultv-at-dispatch-MT-step-contra
    (implies (and (inv MT MA)
                (not (b1p (MT-specultv-at-dispatch?
                           (MT-step MT MA sigs))))
                (not (MT-CMI-p (MT-step MT MA sigs)))
                (not (b1p (flush-all? MA sigs)))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (equal (MT-specultv-at-dispatch? MT) 0))
```

```
  :hints (("goal" :in-theory (enable equal-b1p-converter)))))

(local
(defthm MT-modified-at-dispatch-MT-step-contra
    (implies (and (inv MT MA)
                  (not (b1p (MT-modified-at-dispatch?
                            (MT-step MT MA sigs))))
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (not (b1p (flush-all? MA sigs)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (equal (MT-modified-at-dispatch? MT) 0))
  :hints (("Goal" :in-theory (enable equal-b1p-converter)))))

; This is an important lemma.  If an instruction is dispatched, and
; if it is a register modifier of register idx, the value of the
; signal dispatch-dest-reg from the dispatch logic is idx.
; In other words, dispatch-dest-reg designates the destination register
; of the dispatched instruction.
(defthm not-reg-modifier-if-not-dispatch-dest-reg
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (not (b1p (INST-specultv? (INST-at-stg '(DQ 0) MT))))
                  (not (b1p (INST-modified? (INST-at-stg '(DQ 0) MT))))
                  (not (INST-fetch-error-detected-p (INST-at-stg '(DQ 0) MT)))
                  (not (equal idx (dispatch-dest-reg MA))))
             (not (reg-modifier-p idx (INST-at-stg '(DQ 0) MT))))
  :hints (("Goal" :in-theory (e/d (reg-modifier-p dispatch-dest-reg
                                   DQ-out-dest-reg
                                               dispatch-inst?
                                               INST-DEST-REG
                                               INST-CNTLV
                                               INST-OPCODE
                                               DECODE rdb logbit*
                                               INST-wb?
                                   lift-b-ops)
                                  (DE-VALID-CONSISTENT)))))

; Yet another complicated lemma.  Let us consider register idx.
; Suppose the register reference table has wait? flag set to 0 for
; idx, which suggests no instruction in the ROB is modifying register
; idx.  Further suppose dispatch-dest-reg, an output from the
; dispatching logic, is not idx.  This implies that the dispatched
; instruction does not modify idx, either.  Thus, there is no register
; modifier in the ROB in the next cycle.
(defthm not-LRM-in-ROB-MT-step-if-dispatch-dest-reg-differs
    (implies (and (inv MT MA)
                  (not (b1p (reg-ref-wait?
                            (reg-tbl-nth idx (DQ-reg-tbl (MA-DQ MA))))))
                  (not (equal idx (dispatch-dest-reg MA)))
                  (not (b1p (flush-all? MA sigs)))
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (not (b1p (MT-specultv-at-dispatch?
                            (MT-step MT MA sigs))))
                  (not (b1p (MT-modified-at-dispatch?
                            (MT-step MT MA sigs))))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (rname-p idx))
             (not (exist-LRM-in-ROB-p idx (MT-step MT MA sigs))))
  :hints (("Goal" :cases ((b1p (dispatch-inst? MA))))))

; If the ROB is writing its value to the register file, and I is the
```

```
; instruction at the head of the ROB, then I is committing in this
; cycle.
(defthm INST-commit-LRM-in-ROB
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (b1p (ROB-write-reg? (MA-ROB MA)))
                  (dispatched-p i)
                  (not (committed-p i))
                  (equal (MT-ROB-head MT) (INST-tag i))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-commit? i MA) 1))
  :Hints (("Goal" :in-theory (enable ROB-write-reg? INST-commit?
                                     commit-inst?
                                     INST-commit? equal-b1p-converter
                                     committed-p dispatched-p
                                     lift-b-ops))))

(local
(encapsulate nil
(local
 (defthm committed-p-step-inst-if-following-inst-commit
     (implies (and (inv MT MA)
                   (b1p (INST-commit? i MA))
                   (INST-in i MT) (INST-p i)
                   (INST-in j MT) (INST-p j)
                   (INST-in-order-p j i MT)
                   (MAETT-p MT) (MA-state-p MA))
              (committed-p (step-INST j MT MA sigs)))
   :hints (("Goal" :in-theory (enable INST-commit? lift-b-ops)
                   :use (:instance UNCOMMITTED-INST-P-IS-AFTER-MT-ROB-HEAD
                                   (i j)))))))

; a help lemma
(defthm commit-inst-step-inst-car-if-LRM-commit
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (subtrace-p trace MT) (INST-listp trace)
                  (subtrace-after-p i trace MT)
                  (trace-exist-LRM-in-ROB-p idx trace)
                  (b1p (INST-commit?
                        (trace-LRM-in-ROB idx trace) MA))
                  (MAETT-p MT) (MA-state-p MA))
             (committed-p (step-inst i MT MA sigs))))
))

(encapsulate nil
(local
(defthm LRM-in-ROB-MT-step-if-inst-commit-LRM-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (rname-p idx)
                  (not (b1p (dispatch-inst? MA)))
                  (trace-exist-LRM-in-ROB-p idx trace)
                  (b1p (inst-commit?
                        (trace-LRM-in-ROB idx trace) MA))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (not (trace-exist-LRM-in-ROB-p idx
                        (step-trace trace MT MA sigs ISA spc smc))))
  :hints (("Goal" :restrict
                  ((commit-inst-step-inst-car-if-LRM-commit
                    ((trace (cdr trace)))))))))
```

```
; If the last register modifier in the ROB is committing, and no new
; modifier is dispatched, there will be no register modifier in the
; ROB, because no more modifier is left in the ROB.
(defthm LRM-in-ROB-MT-step-if-inst-commit-LRM
    (implies (and (inv MT MA)
                  (not (b1p (dispatch-inst? MA)))
                  (exist-LRM-in-ROB-p idx MT)
                  (b1p (inst-commit? (LRM-in-ROB idx MT) MA))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (rname-p idx))
             (not (exist-LRM-in-ROB-p idx (MT-step MT MA sigs))))
  :hints (("Goal" :in-theory (enable exist-LRM-in-ROB-p
                                     LRM-in-ROB))))
)

(encapsulate nil
(local
(defthm LRM-in-ROB-MT-step-if-inst-commit-LRM-2-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (not (reg-modifier-p idx
                                       (inst-at-stg-in-trace '(DQ 0) trace)))
                  (trace-exist-LRM-in-ROB-p idx trace)
                  (b1p (inst-commit? (trace-LRM-in-ROB idx trace)
                                     MA))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (rname-p idx))
             (not (trace-exist-LRM-in-ROB-p idx
                       (step-trace trace MT MA sigs ISA spc smc))))
  :hints (("Goal" :restrict
                  ((commit-inst-step-inst-car-if-LRM-commit
                    ((trace (cdr trace)))))))))

; Similar to LRM-in-ROB-MT-step-if-inst-commit-LRM,
; except that there may be a dispatched instruction, but it is not
; modifying register idx.
(defthm LRM-in-ROB-MT-step-if-inst-commit-LRM-2
    (implies (and (inv MT MA)
                  (not (reg-modifier-p idx (inst-at-stg '(DQ 0) MT)))
                  (exist-LRM-in-ROB-p idx MT)
                  (b1p (inst-commit? (LRM-in-ROB idx MT) MA))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (rname-p idx))
             (not (exist-LRM-in-ROB-p idx (MT-step MT MA sigs))))
  :hints (("Goal" :in-theory (enable exist-LRM-in-ROB-p
                                     LRM-in-ROB
                                     INST-at-stg))))
)

; If the last register modifier is not at the head of the ROB, it
; does not commit this cycle.
(defthm not-INST-commit-LRM-in-ROB-if-not-ROB-head
    (implies (and (inv MT MA)
                  (not (equal (MT-ROB-head MT)
                              (INST-tag (LRM-in-ROB idx MT))))
                  (exist-LRM-in-ROB-p idx MT)
                  (MAETT-p MT) (MA-state-p MA) (rname-p idx))
             (equal (INST-commit? (LRM-in-ROB idx MT) MA) 0))
  :hints (("Goal" :in-theory (enable INST-commit? equal-b1p-converter
                                     lift-b-ops))))

; Last register modifier in the ROB is a writeback instruction.
```

```
(defthm INST-wb-LRM-in-ROB
    (implies (exist-LRM-in-ROB-p idx MT)
             (equal (INST-wb? (LRM-in-ROB idx MT)) 1))
  :hints (("Goal" :in-theory (e/d (reg-modifier-p equal-b1p-converter)
                                  (reg-modifier-p-LRM-in-ROB))
                  :use (:instance reg-modifier-p-LRM-in-ROB))))

; The last (general) register modifier in the ROB is not a writeback
; instruction to a special register.
(defthm INST-wb-sreg-LRM-in-ROB
    (implies (exist-LRM-in-ROB-p idx MT)
             (equal (INST-wb-sreg? (LRM-in-ROB idx MT)) 0))
  :hints (("Goal" :in-theory (e/d (reg-modifier-p equal-b1p-converter)
                                  (reg-modifier-p-LRM-in-ROB))
                  :use (:instance reg-modifier-p-LRM-in-ROB))))

; The last special register modifier is a writeback instruction.
(defthm INST-wb-LSRM-in-ROB
    (implies (exist-LSRM-in-ROB-p idx MT)
             (equal (INST-wb? (LSRM-in-ROB idx MT)) 1))
  :hints (("Goal" :in-theory (e/d (sreg-modifier-p equal-b1p-converter)
                                  (sreg-modifier-p-LSRM-in-ROB))
                  :use (:instance sreg-modifier-p-LSRM-in-ROB))))

; The last special register modifier writes back into a special register.
(defthm INST-wb-sreg-LSRM-in-ROB
    (implies (exist-LSRM-in-ROB-p idx MT)
             (equal (INST-wb-sreg? (LSRM-in-ROB idx MT)) 1))
  :hints (("Goal" :in-theory (e/d (sreg-modifier-p equal-b1p-converter)
                                  (sreg-modifier-p-LSRM-in-ROB))
                  :use (:instance sreg-modifier-p-LSRM-in-ROB))))

(encapsulate nil
(local
(defthm ROBE-WB-INST-WB-special
    (implies (and (inv MT MA)
                  (equal (MT-rob-head MT) (INST-tag i))
                  (INST-in i MT) (INST-p i)
                  (not (INST-fetch-error-detected-p i))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (INST-specultv? i)))
                  (dispatched-p i) (not (committed-p i))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (robe-wb? (nth-robe (MT-rob-head MT) (MA-rob MA)))
                    (INST-wb? i))))))

(local
(defthm ROBE-WB-INST-WB-SREG-special
    (implies (and (inv MT MA)
                  (equal (MT-rob-head MT) (INST-tag i))
                  (INST-in i MT) (INST-p i)
                  (not (INST-fetch-error-detected-p i))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (INST-specultv? i)))
                  (dispatched-p i) (not (committed-p i))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (robe-wb-sreg? (nth-robe (MT-rob-head MT) (MA-rob MA)))
                    (INST-wb-sreg? i))))))

(local
 (defthm ROBE-EXCPT-INST-EXCPT-FLAGS-special
    (implies (and (inv MT MA)
```

```
                (equal (MT-ROB-head MT) (INST-tag i))
                (complete-stg-p (INST-stg i))
                (not (b1p (INST-modified? i)))
                (not (b1p (INST-specultv? i)))
                (MAETT-p MT) (MA-state-p MA)
                (INST-in i MT) (INST-p i))
           (equal (robe-excpt (nth-robe (MT-ROB-head MT) (MA-rob MA)))
                  (INST-excpt-flags i)))))

(local
(defthm not-INST-commit-LRM-in-ROB-help
    (implies (and (inv MT MA)
                  (not (b1p (ROB-write-reg? (MA-ROB MA))))
                  (not (b1p (INST-specultv? (LRM-in-ROB idx MT))))
                  (not (b1p (INST-modified? (LRM-in-ROB idx MT))))
                  (not (b1p (INST-excpt? (LRM-in-ROB idx MT))))
                  (exist-LRM-in-ROB-p idx MT)
                  (MAETT-p MT) (MA-state-p MA) (rname-p idx))
             (equal (INST-commit? (LRM-in-ROB idx MT) MA) 0))
  :hints (("Goal" :in-theory
                  (e/d (INST-commit? lift-b-ops
                        INST-excpt? equal-b1p-converter
                        COMMIT-INST?
                        ROB-write-reg?)
                       (uniq-inst-of-tag-if-commit-inst
                        COMPLETE-INST-OF-TAG-IF-COMMIT-INST
                        NOT-EXECUTE-STG-P-INST-AT-ROB-HEAD-IF-COMMIT-INST))))))

; When register modifier commits, ROB-write-reg? must be turned on.
; Note: The rule is written in the form of contrapositive.
(defthm not-INST-commit-LRM-in-ROB
    (implies (and (inv MT MA)
                  (not (b1p (ROB-write-reg? (MA-ROB MA))))
                  (not (b1p (MT-specultv-at-dispatch? (MT-STEP MT MA SIGS))))
                  (not (b1p (MT-modified-at-dispatch? (MT-STEP MT MA SIGS))))
                  (not (MT-CMI-p (MT-step MT MA SIGS)))
                  (not (b1p (flush-all? MA SIGS)))
                  (exist-LRM-in-ROB-p idx MT)
                  (MAETT-p MT) (MA-state-p MA) (rname-p idx))
             (equal (INST-commit? (LRM-in-ROB idx MT) MA) 0))
  :hints (("Goal" :in-theory (enable equal-b1p-converter
                                     flush-all? lift-b-ops)
                  :restrict ((MT-CMI-P-IF-MODIFIED-INST-COMMIT
                              ((i (LRM-in-ROB idx MT)))))
                  :cases ((b1p (INST-specultv?
                                (LRM-in-ROB idx MT)))
                          (b1p (INST-modified?
                                (LRM-in-ROB idx MT)))))
          ("subgoal 3" :cases ((b1p (INST-excpt?
                                     (LRM-in-ROB idx MT)))))
          ("subgoal 3.1" :cases
                 ((complete-stg-p (INST-stg (LRM-IN-ROB IDX MT)))))))
)

(encapsulate nil
(local
 (defthm ROBE-dest-INST-dest-reg-special
    (implies (and (inv MT MA)
                  (equal (MT-ROB-head MT) (INST-tag i))
                  (INST-writeback-p i)
                  (complete-stg-p (INST-stg i))
                  (not (b1p (INST-modified? i)))
```

```
                (not (b1p (INST-specultv? i)))
                (not (INST-fetch-error-detected-p i))
                (MAETT-p MT) (MA-state-p MA)
                (INST-in i MT) (INST-p i))
           (equal (robe-dest (nth-robe (MT-ROB-head MT) (MA-rob MA)))
                  (INST-dest-reg i)))))

(local
(defthm not-INST-commit-LRM-in-ROB-if-rob-write-rid-differs-local
    (implies (and (inv MT MA)
                (not (equal (rob-write-rid (MA-ROB MA)) idx))
                (exist-LRM-in-ROB-p idx MT)
                (not (b1p (INST-specultv? (LRM-in-ROB idx MT))))
                (not (b1p (INST-modified? (LRM-in-ROB idx MT))))
                (not (b1p (INST-excpt? (LRM-in-ROB idx MT))))
                (MAETT-p MT) (MA-state-p MA) (rname-p idx))
           (equal (INST-commit? (LRM-in-ROB idx MT) MA) 0))
  :hints (("Goal" :in-theory (enable rob-write-rid INST-commit? lift-b-ops
                                     equal-b1p-converter INST-excpt?)))))

; If the last modifier of register idx in the ROB is committing,
; signal rob-write-rid from the ROB designates the modified register
; idx.
(defthm not-INST-commit-LRM-in-ROB-if-rob-write-rid-differs
    (implies (and (inv MT MA)
                (not (equal (rob-write-rid (MA-ROB MA)) idx))
                (exist-LRM-in-ROB-p idx MT)
                (not (b1p (flush-all? MA sigs)))
                (not (b1p (MT-specultv-at-dispatch?
                           (MT-step MT MA sigs))))
                (not (b1p (MT-modified-at-dispatch?
                           (MT-step MT MA sigs))))
                (not (MT-CMI-p (MT-step MT MA sigs)))
                (MAETT-p MT) (MA-state-p MA) (rname-p idx))
           (equal (INST-commit? (LRM-in-ROB idx MT) MA) 0))
  :hints (("Goal" :in-theory (enable equal-b1p-converter
                                     flush-all? lift-b-ops)
                :restrict ((MT-CMI-P-IF-MODIFIED-INST-COMMIT
                            ((i (LRM-in-ROB idx MT)))))
                :cases ((b1p (INST-specultv?
                              (LRM-in-ROB idx MT)))
                        (b1p (INST-modified?
                              (LRM-in-ROB idx MT)))))
          ("subgoal 3" :cases ((b1p (INST-excpt?
                                     (LRM-in-ROB idx MT)))))
          ("subgoal 3.1" :cases
                 ((complete-stg-p (INST-stg (LRM-IN-ROB IDX MT)))))))
)

(encapsulate nil
(local
(defthm LRM-in-ROB-MT-step-if-inst-commit-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (trace-exist-LRM-in-ROB-p idx trace)
                (not (b1p (flush-all? MA sigs)))
                (not (b1p (inst-commit?
                           (trace-LRM-in-ROB idx trace)
                           MA)))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (trace-exist-LRM-in-ROB-p idx
                     (step-trace trace MT MA sigs ISA spc smc)))))
```

```
; If the last register modifier in the ROB does not commit, the ROB
; will continue to hold the register modifier.
(defthm LRM-in-ROB-MT-step-if-inst-commit
    (implies (and (inv MT MA)
                  (exist-LRM-in-ROB-p idx MT)
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (inst-commit? (LRM-in-ROB idx MT) MA)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (exist-LRM-in-ROB-p idx (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable exist-LRM-in-ROB-p
                                     LRM-in-ROB))))
)

(encapsulate nil
(local
(defthm LRM-in-ROB-if-dispatch-inst-help
    (implies (and (inv MT MA)
                  (b1p (dispatch-inst? MA))
                  (b1p (cntlv-wb? (dispatch-cntlv MA)))
                  (not (b1p (cntlv-wb-sreg? (dispatch-cntlv MA))))
                  (not (b1p (INST-specultv? (INST-at-stg '(DQ 0) MT))))
                  (not (b1p (INST-modified? (INST-at-stg '(DQ 0) MT))))
                  (not (INST-fetch-error-detected-p (INST-at-stg '(DQ 0) MT)))
                  (MAETT-p MT) (MA-state-p MA))
             (reg-modifier-p (dispatch-dest-reg MA)
                             (INST-at-stg '(DQ 0) MT)))
  :hints (("Goal" :in-theory (enable reg-modifier-p DQ-OUT-DEST-REG
                                     dispatch-cntlv INST-WB? INST-wb-sreg?
                                     INST-dest-reg INST-cntlv INST-opcode
                                     lift-b-ops decode rdb logbit*
                                     dispatch-dest-reg)))))

; If the dispatched instruction is a write-back instruction,
; then the dispatched instruction is a modifier of the register
; that is designated by signal dispatch-dest-reg from the dispatch logic.
(defthm LRM-in-ROB-if-dispatch-inst
    (implies (and (inv MT MA)
                  (b1p (dispatch-inst? MA))
                  (b1p (cntlv-wb? (dispatch-cntlv MA)))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (cntlv-wb-sreg? (dispatch-cntlv MA))))
                  (not (b1p (MT-specultv-at-dispatch?
                             (MT-step MT MA sigs))))
                  (not (b1p (MT-modified-at-dispatch?
                             (MT-step MT MA sigs))))
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (reg-modifier-p (dispatch-dest-reg MA)
                             (INST-at-stg '(DQ 0) MT))))
)

(encapsulate nil
(local
(defthm LRM-in-ROB-MT-step-if-dispatch-induct
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (member-equal i trace) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (dispatch-inst? MA))
                  (equal (INST-stg i) '(DQ 0))
                  (reg-modifier-p idx i)
```

```
                (MAETT-p MT) (MA-state-p MA))
           (trace-exist-LRM-in-ROB-p idx
                     (step-trace trace MT MA sigs ISA spc smc)))))

(local
(defthm LRM-in-ROB-MT-step-if-dispatch-help
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (dispatch-inst? MA))
                  (equal (INST-stg i) '(DQ 0))
                  (reg-modifier-p idx i)
                  (MAETT-p MT) (MA-state-p MA))
             (exist-LRM-in-ROB-p idx (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable exist-LRM-in-ROB-p INST-in)))))

; If the dispatched instruction is a register modifier, then
; the ROB contains a register modifier in the next clock cycle.
(defthm LRM-in-ROB-MT-step-if-dispatch
    (implies (and (inv MT MA)
                  (b1p (dispatch-inst? MA))
                  (not (b1p (flush-all? MA sigs)))
                  (reg-modifier-p idx (INST-at-stg '(DQ 0) MT))
                  (MAETT-p MT) (MA-state-p MA))
             (exist-LRM-in-ROB-p idx (MT-step MT MA sigs)))
  :hints (("Goal" :restrict ((LRM-in-ROB-MT-step-if-dispatch-help
                               ((i (INST-at-stg '(DQ 0) MT))))))))
)

; A landmark lemma.
; Following two lemmas imply that the wait? bit of register reference table
; in the next cycle is correct.  Register reference table contains
; 0 for wait? bit, then there is no register modifier in the ROB.
; Conversely, if wait? bit is 1, there is a register modifier.
(defthm not-LRM-in-ROB-MT-step-if-not-reg-ref-wait
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (rname-p idx)
                  (not (b1p (MT-specultv-at-dispatch?
                              (MT-step MT MA sigs))))
                  (not (b1p (MT-modified-at-dispatch?
                              (MT-step MT MA sigs))))
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (not (b1p (reg-ref-wait?
                              (reg-tbl-nth idx
                                           (DQ-reg-tbl (step-DQ MA sigs)))))))
             (not (exist-LRM-in-ROB-p idx (MT-step MT MA sigs))))
  :hints (("Goal" :in-theory (enable STEP-REG-REF lift-b-ops)
                  :cases ((b1p (flush-all? MA sigs))))
          (use-hint-always (:cases ((b1p (dispatch-inst? MA)))))
          (use-hint-always
           (:cases ((b1p (reg-ref-wait?
                          (reg-tbl-nth (rob-write-rid (MA-rob MA))
                                       (DQ-reg-tbl (MA-DQ MA))))))))))

; A landmark lemma. See the comment above.
(defthm LRM-in-ROB-MT-step-if-reg-ref-wait
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (rname-p idx)
                  (not (b1p (MT-specultv-at-dispatch?
                              (MT-step MT MA sigs))))
```

```
                (not (b1p (MT-modified-at-dispatch?
                           (MT-step MT MA sigs))))
                (not (MT-CMI-p (MT-step MT MA sigs)))
                (b1p (reg-ref-wait?
                      (reg-tbl-nth idx
                                   (DQ-reg-tbl (step-DQ MA sigs))))))
           (exist-LRM-in-ROB-p idx (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable STEP-REG-REF lift-b-ops))))

(encapsulate nil
(local
(defthm trace-LRM-in-ROB-step-trace-if-dispatch-inst
    (implies (and (inv MT MA)
                (member-equal i trace) (INST-p i)
                (subtrace-p trace MT) (INST-listp trace)
                (not (b1p (flush-all? MA sigs)))
                (b1p (dispatch-inst? MA))
                (equal (INST-stg i) '(DQ 0))
                (reg-modifier-p idx i)
                (MAETT-p MT) (MA-state-p MA))
           (trace-exist-LRM-in-ROB-p idx
                       (step-trace trace MT MA sigs ISA spc smc)))))

(local
(defthm LRM-in-ROB-MT-step-if-dispatch-inst-induct
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (reg-modifier-p idx i)
                (member-equal i trace) (INST-p i)
                (not (b1p (flush-all? MA sigs)))
                (b1p (dispatch-inst? MA))
                (equal (INST-stg i) '(DQ 0))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (equal (trace-LRM-in-ROB idx
                        (step-trace trace MT MA sigs ISA spc smc))
                  (step-INST i MT MA sigs)))
  :hints (("goal" :restrict
                ((NOT-TRACE-LRM-IN-ROB-STEP-TRACE
                  ((i (car trace)))))))))

(local
(defthm LRM-in-ROB-MT-step-if-dispatch-inst-help
    (implies (and (inv MT MA)
                (reg-modifier-p idx i)
                (INST-in i MT) (INST-p i)
                (not (b1p (flush-all? MA sigs)))
                (b1p (dispatch-inst? MA))
                (equal (INST-stg i) '(DQ 0))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (equal (LRM-in-ROB idx (MT-step MT MA sigs))
                  (step-INST i MT MA sigs)))
  :hints (("Goal" :in-theory (enable LRM-in-ROB INST-in)))))

; If instruction i at DQ0 is a register modifier of register idx, and
; it is dispatched in this cycle, i is the last register modifier in
; the ROB in the next cycle.
(defthm LRM-in-ROB-MT-step-if-dispatch-inst
    (implies (and (inv MT MA)
                (b1p (dispatch-inst? MA))
                (not (b1p (flush-all? MA sigs)))
                (reg-modifier-p idx (INST-at-stg '(DQ 0) MT))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
```

```
            (equal (LRM-in-ROB idx (MT-step MT MA sigs))
                   (step-INST (INST-at-stg '(DQ 0) MT) MT MA sigs)))
  :hints (("Goal" :restrict
                  ((LRM-in-ROB-MT-step-if-dispatch-inst-help
                    ((i (INST-at-stg '(DQ 0) MT))))))))
)

(local
(defthm trace-LRM-in-ROB-step-trace-if-not-INST-commit
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (trace-exist-LRM-in-ROB-p idx trace)
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (INST-commit?
                             (trace-LRM-in-ROB idx trace) MA)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (trace-exist-LRM-in-ROB-p idx
                (step-trace trace MT MA sigs ISA spc smc)))))

(encapsulate nil
(local
(defthm LRM-in-ROB-MT-step-if-not-dispatch-inst-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (not (b1p (dispatch-inst? MA)))
                  (not (b1p (flush-all? MA sigs)))
                  (trace-exist-LRM-in-ROB-p idx trace)
                  (not (b1p (INST-commit?
                             (trace-LRM-in-ROB idx trace) MA)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (rname-p idx))
             (equal (trace-LRM-in-ROB idx
                           (step-trace trace MT MA sigs ISA spc smc))
                    (step-INST (trace-LRM-in-ROB idx trace)
                               MT MA sigs)))))

; help lemma to prove LRM-in-ROB-MT-step
(defthm LRM-in-ROB-MT-step-if-not-dispatch-inst
    (implies (and (inv MT MA)
                  (not (b1p (dispatch-inst? MA)))
                  (not (b1p (flush-all? MA sigs)))
                  (exist-LRM-in-ROB-p idx MT)
                  (not (b1p (INST-commit? (LRM-in-ROB idx MT) MA)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (rname-p idx))
             (equal (LRM-in-ROB idx (MT-step MT MA sigs))
                    (step-INST (LRM-in-ROB idx MT) MT MA sigs)))
  :hints (("Goal" :in-theory (enable LRM-in-ROB
                                     exist-LRM-in-ROB-p))))

)

(encapsulate nil
(local
(defthm LRM-in-ROB-MT-step-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (not (b1p (flush-all? MA sigs)))
                  (not (reg-modifier-p idx
                                       (INST-at-stg-in-trace '(DQ 0) trace)))
                  (trace-exist-LRM-in-ROB-p idx trace)
                  (not (b1p (INST-commit? (trace-LRM-in-ROB idx trace) MA)))
```

```
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                (rname-p idx))
           (equal (trace-LRM-in-ROB idx
                      (step-trace trace MT MA sigs ISA spc smc))
             (step-INST (trace-LRM-in-ROB idx trace)
                        MT MA sigs)))))

; A landmark lemma.
; If the last register modifier in the ROB does not commit in this
; cycle, and the instruction at DQ0 is not a modifier, then the
; current last register modifier will continue to be the last register
; modifier in the next cycle.
(defthm LRM-in-ROB-MT-step
    (implies (and (inv MT MA)
                (not (b1p (flush-all? MA sigs)))
                (not (reg-modifier-p idx (INST-at-stg '(DQ 0) MT)))
                (exist-LRM-in-ROB-p idx MT)
                (not (b1p (INST-commit? (LRM-in-ROB idx MT) MA)))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                (rname-p idx))
           (equal (LRM-in-ROB idx (MT-step MT MA sigs))
             (step-INST (LRM-in-ROB idx MT) MT MA sigs)))
  :hints (("Goal" :in-theory (enable LRM-in-ROB
                                     INST-at-stg
                                     exist-LRM-in-ROB-p))))
)

; Lemmas for the proof of INST-tag-LRM-MT-step
;
; The proof of INST-tag-LRM-MT-step is divided into three cases, where
; dispatch-inst? is false, where the instruction at DQ0 is not a
; register modifier, and where both conditions are false.  In either
; case, the new value of robe field of the register reference table is
; the correct Tomasulo's tag for the last register modifier in the
; ROB.
(defthm INST-tag-LRM-MT-step-if-not-dispatch-inst
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                (rname-p idx)
                (not (b1p (dispatch-inst? MA)))
                (not (b1p (MT-specultv-at-dispatch? (MT-step MT MA sigs))))
                (not (b1p (MT-modified-at-dispatch? (MT-step MT MA sigs))))
                (not (MT-CMI-p (MT-step MT MA sigs)))
                (b1p (reg-ref-wait?
                      (step-reg-ref (reg-tbl-nth idx (DQ-reg-tbl (MA-DQ MA)))
                                    idx MA sigs))))
           (equal (INST-tag (LRM-in-ROB
                             idx (MT-step MT MA sigs)))
             (reg-ref-tag (step-reg-ref
                           (reg-tbl-nth idx (DQ-reg-tbl (MA-DQ MA)))
                           idx MA sigs))))
  :hints (("Goal" :in-theory (enable step-reg-ref lift-b-ops))))

(defthm INST-tag-LRM-MT-step-if-not-reg-modifier
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                (rname-p idx)
                (not (reg-modifier-p idx (INST-at-stg '(DQ 0) MT)))
                (not (b1p (MT-specultv-at-dispatch? (MT-step MT MA sigs))))
                (not (b1p (MT-modified-at-dispatch? (MT-step MT MA sigs))))
                (not (MT-CMI-p (MT-step MT MA sigs)))
                (b1p (reg-ref-wait?
```

```
                        (step-reg-ref (reg-tbl-nth idx (DQ-reg-tbl (MA-DQ MA)))
                                      idx MA sigs))))
             (equal (INST-tag (LRM-in-ROB
                               idx (MT-step MT MA sigs)))
                    (reg-ref-tag (step-reg-ref
                                  (reg-tbl-nth idx (DQ-reg-tbl (MA-DQ MA)))
                                  idx MA sigs))))
    :hints (("Goal" :in-theory (enable step-reg-ref lift-b-ops))))

(defthm INST-tag-LRM-MT-step-if-dispatch-inst
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (rname-p idx)
                  (b1p (dispatch-inst? MA))
                  (reg-modifier-p idx (INST-at-stg '(DQ 0) MT))
                  (not (b1p (MT-specultv-at-dispatch? (MT-step MT MA sigs))))
                  (not (b1p (MT-modified-at-dispatch? (MT-step MT MA sigs))))
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (b1p (reg-ref-wait?
                        (step-reg-ref (reg-tbl-nth idx (DQ-reg-tbl (MA-DQ MA)))
                                      idx MA sigs))))
             (equal (reg-ref-tag (step-reg-ref
                                  (reg-tbl-nth idx (DQ-reg-tbl (MA-DQ MA)))
                                  idx MA sigs))
                    (INST-tag (LRM-in-ROB
                               idx (MT-step MT MA sigs)))))
  :hints (("Goal" :in-theory (enable step-reg-ref lift-b-ops))))

; Combining the three lemmas above, we conclude that the
; Tomasulo's tag for the last register modifier in the ROB is
; in the register reference table in the next cycle.
(defthm INST-tag-LRM-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (rname-p idx)
                  (not (b1p (MT-specultv-at-dispatch? (MT-step MT MA sigs))))
                  (not (b1p (MT-modified-at-dispatch? (MT-step MT MA sigs))))
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (b1p (reg-ref-wait?
                        (step-reg-ref (reg-tbl-nth idx (DQ-reg-tbl (MA-DQ MA)))
                                      idx MA sigs))))
             (equal (INST-tag (LRM-in-ROB idx (MT-step MT MA sigs)))
                    (reg-ref-tag (step-reg-ref
                                  (reg-tbl-nth idx (DQ-reg-tbl (MA-DQ MA)))
                                  idx MA sigs))))
  :hints (("Goal" :cases ((not (b1p (dispatch-inst? MA)))
                          (not (reg-modifier-p idx
                                               (INST-at-stg '(DQ 0) MT)))))))

; Consistent-reg-ref-p is an invariant for register idx.
(defthm consistent-reg-ref-p-MT-step
    (implies (and (inv MT MA)
                  (rname-p idx)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (not (MT-CMI-p (MT-step MT MA sigs))))
             (consistent-reg-ref-p idx (MT-step MT MA sigs)
                                   (MA-step MA sigs)))
  :hints (("Goal" :in-theory (enable consistent-reg-ref-p))))

(encapsulate nil
(local
(defthm consistent-reg-tbl-p-preserved-help
```

```
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (integerp idx) (<= 0 idx) (<= idx *num-regs*)
                  (not (MT-CMI-p (MT-step MT MA sigs))))
             (consistent-reg-tbl-under idx (MT-step MT MA sigs)
                                       (MA-step MA sigs)))
  :hints (("Goal" :in-theory (enable consistent-reg-tbl-p rname-p
                                     unsigned-byte-p)))))

; Consistent-reg-tbl-p is an invariant.
(defthm consistent-reg-tbl-p-preserved
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (not (MT-CMI-p (MT-step MT MA sigs))))
             (consistent-reg-tbl-p (MT-step MT MA sigs)
                                   (MA-step MA sigs)))
  :hints (("Goal" :in-theory (enable consistent-reg-tbl-p))))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Proof of consistent-sreg-tbl-p-preserved
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Proof of consistent-sreg-tbl-p for initial states
(defthm consistent-sreg-tbl-p-init-MT
    (implies (and (MA-state-p MA) (b1p (MA-flushed? MA)))
             (consistent-sreg-tbl-p (init-MT MA) MA))
  :hints (("goal" :in-theory (enable consistent-sreg-tbl-p
                                     consistent-sreg-ref-p init-MT
                                     MA-flushed? lift-b-ops DQ-empty?
                                     SREG-TBL-EMPTY?
                                     exist-LSRM-in-ROB-p))))

;;; Invariant proof
; This is a lemma to open up the definition of step-DQ.
(defthm sreg-tbl-nth-step-DQ
    (implies (and (MA-state-p MA) (srname-p idx))
             (equal (sreg-tbl-nth idx (DQ-sreg-tbl (step-DQ MA sigs)))
                    (step-sreg-ref (sreg-tbl-nth idx (DQ-sreg-tbl (MA-DQ MA)))
                                   idx MA sigs)))
  :hints (("goal" :in-theory (enable step-DQ step-sreg-tbl unsigned-byte-p
                                     srname-p sreg-tbl-nth))))

;;;; Proof of not-LSRM-in-ROB-MT-step-if-not-reg-ref-wait
(encapsulate nil
(local
(defthm dispatched-p-LSRM-in-ROB-help
    (implies (trace-exist-LSRM-in-ROB-p idx trace)
             (dispatched-p (trace-LSRM-in-ROB idx trace)))))

; The (special) register modifier in the ROB is dispatched.
(defthm dispatched-p-LSRM-in-ROB
    (implies (exist-LSRM-in-ROB-p idx MT)
             (dispatched-p (LSRM-in-ROB idx MT)))
  :hints (("goal" :in-theory (enable exist-LSRM-in-ROB-p
                                     LSRM-in-ROB))))
)

(encapsulate nil
(defthm not-committed-p-LSRM-in-ROB-help
    (implies (trace-exist-LSRM-in-ROB-p idx trace)
             (not (committed-p (trace-LSRM-in-ROB idx trace)))))
```

```
; The (special) register modifier in the ROB is not committed.
(defthm not-committed-p-LSRM-in-ROB
    (implies (exist-LSRM-in-ROB-p idx MT)
             (not (committed-p (LSRM-in-ROB idx MT))))
  :hints (("Goal" :in-theory (enable exist-LSRM-in-ROB-p
                                     LSRM-in-ROB))))
)

; The instruction at the head of the ROB commits when ROB-write-sreg?
; is 1.
(defthm INST-commit-LSRM-in-ROB
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (b1p (ROB-write-sreg? (MA-ROB MA)))
                  (dispatched-p i)
                  (not (committed-p i))
                  (equal (MT-ROB-head MT) (INST-tag i))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-commit? i MA) 1))
  :Hints (("Goal" :in-theory (enable ROB-write-sreg? INST-commit?
                                     commit-inst?
                                     INST-commit? equal-b1p-converter
                                     committed-p dispatched-p
                                     lift-b-ops))))

; If the dispatched instruction is a special register modifier,
; dispatch-dest-reg designates the modified special register.
(defthm not-sreg-modifier-if-not-dispatch-dest-reg
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (not (b1p (INST-specultv? (INST-at-stg '(DQ 0) MT))))
                  (not (b1p (INST-modified? (INST-at-stg '(DQ 0) MT))))
                  (not (INST-fetch-error-detected-p (INST-at-stg '(DQ 0) MT)))
                  (not (equal idx (dispatch-dest-reg MA))))
             (not (sreg-modifier-p idx (INST-at-stg '(DQ 0) MT))))
  :hints (("Goal" :in-theory (e/d (sreg-modifier-p dispatch-dest-reg
                                     DQ-out-dest-reg
                                     dispatch-inst?
                                     INST-DEST-REG
                                     INST-CNTLV
                                     INST-OPCODE
                                     DECODE rdb logbit*
                                     INST-wb?
                                     lift-b-ops)
                                   (DE-VALID-CONSISTENT)))))

;  If the dispatched instruction is a special register modifier,
; the wb? and wb-sreg? fields of the dispatch-cntlv signal
; are 1.
(defthm not-sreg-modifier-if-not-cntlv-wb
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (not (b1p (INST-specultv? (INST-at-stg '(DQ 0) MT))))
                  (not (b1p (INST-modified? (INST-at-stg '(DQ 0) MT))))
                  (not (INST-fetch-error-detected-p (INST-at-stg '(DQ 0) MT)))
                  (or (not (b1p (cntlv-wb? (dispatch-cntlv MA))))
                      (not (b1p (cntlv-wb-sreg? (dispatch-cntlv MA))))))
             (not (sreg-modifier-p idx (INST-at-stg '(DQ 0) MT))))
  :hints (("Goal" :in-theory (enable dispatch-cntlv sreg-modifier-p
                                     dispatch-inst? DISPATCH-NO-UNIT?
```

```
                                    dispatch-to-IU? dispatch-to-MU?
                                    dispatch-to-BU? dispatch-to-LSU?
                                    DQ-READY-NO-UNIT? DQ-READY-TO-BU?
                                    DQ-READY-TO-IU? DQ-READY-TO-LSU?
                                    DQ-READY-TO-MU?
                                    INST-WB-SREG? INST-WB?
                                    lift-b-ops))))

(encapsulate nil
(local
(defthm not-LSRM-in-ROB-MT-step-if-flush-all-help
    (implies (and (inv MT MA)
                  (MT-all-commit-before-trace trace MT)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (flush-all? MA sigs)))
             (not (trace-exist-LSRM-in-ROB-p idx
                                             (step-trace trace MT MA sigs
                                                         ISA spc smc))))
  :hints (("goal" :in-theory (enable committed-p))
          (when-found (MT-ALL-COMMIT-BEFORE-TRACE (CDR TRACE) MT)
                      (:cases ((committed-p (car trace))))))))

; No register modifier will be in the MAETT after a flush-all.
(defthm not-LSRM-in-ROB-MT-step-if-flush-all
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (flush-all? MA sigs)))
             (not (exist-LSRM-in-ROB-p idx (MT-step MT MA sigs))))
  :hints (("Goal" :in-theory (enable exist-LSRM-in-ROB-p))))
)

(encapsulate nil
(local
 (defthm local-help
     (implies (and (inv MT MA)
                   (INST-in i MT) (INST-p i)
                   (INST-in j MT) (INST-p j)
                   (INST-in-order-p i j MT)
                   (not (equal i j))
                   (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                   (DQ-stg-p (INST-stg i)) (DQ-stg-p (INST-stg j)))
              (DQ-stg-p (INST-stg (step-INST j MT MA sigs))))
 :hints (("Goal" :in-theory (enable DQ-stg-p step-inst-dq-inst
                                    step-inst-low-level-functions)
                 :use ((:instance uniq-stage-inst)
                       (:instance DQ-stg-index-monotonic))))))

(local
 (defthm not-dispatched-inst-step-inst-if-earlier-non-dispatched
     (implies (and (inv MT MA)
                   (INST-in i MT) (INST-p i)
                   (INST-in j MT) (INST-p j)
                   (INST-in-order-p i j MT)
                   (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                   (not (dispatched-p i)))
              (not (dispatched-p (step-inst j MT MA sigs))))
   :hints (("Goal" :in-theory (e/d (dispatched-p* IFU-IS-LAST-INST)
                                   (inst-is-at-one-of-the-stages))
                   :restrict ((local-help ((i i))))
                   :use (:instance inst-is-at-one-of-the-stages
                         (i j))))
```

```
           (when-found (DQ-stg-p (INST-stg j))
                       (:cases ((equal i j)))))))

; A help lemma
(defthm not-trace-LSRM-in-ROB-step-trace
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (INST-in i MT) (INST-p i)
                  (not (dispatched-p i))
                  (subtrace-after-p i trace MT)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (not (trace-exist-LSRM-in-ROB-p idx
                        (step-trace trace MT MA sigs ISA spc smc)))))
)

(local
(defthm not-trace-LSRM-in-ROB-MT-step-if-no-dispatch
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (srname-p idx)
                  (not (trace-exist-LSRM-in-ROB-p idx trace))
                  (not (b1p (dispatch-inst? MA))))
             (not (trace-exist-LSRM-in-ROB-p idx
                        (step-trace trace MT MA sigs ISA spc smc))))))

(local
(defthm not-trace-LSRM-in-ROB-MT-step-if-not-sreg-modifier
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (srname-p idx)
                  (not (trace-exist-LSRM-in-ROB-p idx trace))
                  (not (sreg-modifier-p idx (INST-at-stg-in-trace '(DQ 0)
                                                                  trace))))
             (not (trace-exist-LSRM-in-ROB-p idx
                           (step-trace trace MT MA sigs ISA spc smc))))
  :hints (("Goal" :in-theory (enable dispatched-p)
                  :restrict ((not-trace-LSRM-in-ROB-step-trace
                               ((i (car trace)))))))))

; If there is no special register modifier in the ROB in the current cycle,
; and the (possibly) dispatched instruction is not a modifier, then
; there will be no modifier in the ROB in the next cycle.
(defthm not-LSRM-in-ROB-MT-step-if-not-sreg-modifier
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (srname-p idx)
                  (not (exist-LSRM-in-ROB-p idx MT))
                  (not (sreg-modifier-p idx (INST-at-stg '(DQ 0) MT))))
             (not (exist-LSRM-in-ROB-p idx (MT-step MT MA sigs))))
  :hints (("Goal" :in-theory (enable exist-LSRM-in-ROB-p
                                     INST-at-stg))))

(local
(encapsulate nil
(local
 (defthm committed-p-step-inst-if-following-inst-commit
     (implies (and (inv MT MA)
                   (b1p (INST-commit? i MA))
                   (INST-in i MT) (INST-p i)
                   (INST-in j MT) (INST-p j)
```

```
                  (INST-in-order-p j i MT)
                  (MAETT-p MT) (MA-state-p MA))
             (committed-p (step-INST j MT MA sigs)))
   :hints (("Goal" :in-theory (enable INST-commit? lift-b-ops)
                   :use (:instance UNCOMMITTED-INST-P-IS-AFTER-MT-ROB-HEAD
                                   (i j))))))

; A local help lemma.
(defthm commit-inst-step-inst-car-if-LSRM-commit
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (subtrace-p trace MT) (INST-listp trace)
                  (subtrace-after-p i trace MT)
                  (trace-exist-LSRM-in-ROB-p idx trace)
                  (b1p (INST-commit?
                        (trace-LSRM-in-ROB idx trace) MA))
                  (MAETT-p MT) (MA-state-p MA))
             (committed-p (step-inst i MT MA sigs))))
))

(encapsulate nil
(local
(defthm LSRM-in-ROB-MT-step-if-inst-commit-LSRM-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (srname-p idx)
                  (not (b1p (dispatch-inst? MA)))
                  (trace-exist-LSRM-in-ROB-p idx trace)
                  (b1p (inst-commit?
                        (trace-LSRM-in-ROB idx trace) MA))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (not (trace-exist-LSRM-in-ROB-p idx
                        (step-trace trace MT MA sigs ISA spc smc))))
  :hints (("Goal" :restrict
                  ((commit-inst-step-inst-car-if-LSRM-commit
                    ((trace (cdr trace)))))))))

; If the last special register modifier commits in this cycle, and no
; instruction is dispatched, then there will be no modifier in the ROB
; next cycle.
(defthm LSRM-in-ROB-MT-step-if-inst-commit-LSRM
    (implies (and (inv MT MA)
                  (not (b1p (dispatch-inst? MA)))
                  (exist-LSRM-in-ROB-p idx MT)
                  (b1p (inst-commit? (LSRM-in-ROB idx MT) MA))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (srname-p idx))
             (not (exist-LSRM-in-ROB-p idx (MT-step MT MA sigs))))
  :hints (("Goal" :in-theory (enable exist-LSRM-in-ROB-p
                                     LSRM-in-ROB))))
)

(encapsulate nil
(local
(defthm LSRM-in-ROB-MT-step-if-inst-commit-LSRM-2-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (not (sreg-modifier-p idx
                                        (inst-at-stg-in-trace '(DQ 0) trace)))
                  (trace-exist-LSRM-in-ROB-p idx trace)
                  (b1p (inst-commit?
                        (trace-LSRM-in-ROB idx trace) MA))
```

```
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                (srname-p idx))
           (not (trace-exist-LSRM-in-ROB-p idx
                     (step-trace trace MT MA sigs ISA spc smc))))
  :hints (("Goal" :restrict
                ((commit-inst-step-inst-car-if-LSRM-commit
                  ((trace (cdr trace)))))))))

; If the last special register modifier commits in this cycle,
; and the dispatched instruction is not a modifier, then
; there will be no register modifier in the ROB next cycle.
(defthm LSRM-in-ROB-MT-step-if-inst-commit-LSRM-2
    (implies (and (inv MT MA)
                (not (sreg-modifier-p idx (inst-at-stg '(DQ 0) MT)))
                (exist-LSRM-in-ROB-p idx MT)
                (b1p (inst-commit? (LSRM-in-ROB idx MT) MA))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                (srname-p idx))
           (not (exist-LSRM-in-ROB-p idx (MT-step MT MA sigs))))
  :hints (("Goal" :in-theory (enable exist-LSRM-in-ROB-p
                                     LSRM-in-ROB
                                     INST-at-stg))))
)

; If there is no sreg modifier in rob, and if no instruction
; is dispatched, there won't be any sreg-modifier in ROB.
(defthm not-LSRM-in-ROB-MT-step-if-no-dispatch
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                (srname-p idx)
                (not (exist-LSRM-in-ROB-p idx MT))
                (not (b1p (dispatch-inst? MA))))
           (not (exist-LSRM-in-ROB-p idx (MT-step MT MA sigs))))
  :hints (("Goal" :in-theory (enable exist-LSRM-in-ROB-p MT-step))))

; The destination special register should be 0 or 1, otherwise an
; illegal instruction is raised.
(defthm INST-dest-reg-if-INST-wb-sreg
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (not (b1p (INST-decode-error? i)))
                (b1p (INST-wb-sreg? i))
                (MAETT-p MT) (MA-state-p MA))
           (< (INST-dest-reg i) 2))
  :hints (("Goal" :in-theory (enable INST-decode-error? lift-b-ops
                                     DECODE-ILLEGAL-INST?
                                     INST-WB-SREG? decode rdb logbit*
                                     INST-CNTLV INST-opcode
                                     INST-DEST-REG INST-rc))))

; Signal ROB-write-rid designates the existing special register
; when a special register is written into.
(defthm srname-p-rob-write-rid
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (INST-modified?
                           (inst-of-tag (MT-rob-head MT) MT))))
                (not (b1p (INST-specultv?
                           (inst-of-tag (MT-rob-head MT) MT))))
                (not (INST-fetch-error-detected-p
                      (inst-of-tag (MT-rob-head MT) MT)))
                (b1p (ROB-write-sreg? (MA-rob MA))))
```

```
            (srname-p (ROB-write-rid (MA-ROB MA))))
  :hints (("goal" :in-theory (enable ROB-write-sreg? srname-p ROB-write-rid
                                     lift-b-ops exception-relations
                                     INST-EXCPT-DETECTED-P)
                  :cases ((b1p (INST-fetch-error?
                                (inst-of-tag (MT-rob-head MT) MT)))
                          (not (b1p (INST-decode-error?
                                     (inst-of-tag (MT-rob-head MT) MT))))))))

; A landmark lemma
; If the wait? bit of the special register reference table is
; 0, there will not be any special register modifier in the ROB.
(defthm not-LSRM-in-ROB-MT-step-if-not-reg-ref-wait
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (srname-p idx)
                  (not (b1p (MT-specultv-at-dispatch?
                             (MT-step MT MA sigs))))
                  (not (b1p (MT-modified-at-dispatch?
                             (MT-step MT MA sigs))))
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (not (b1p (reg-ref-wait?
                             (sreg-tbl-nth idx
                                           (DQ-sreg-tbl (step-DQ MA sigs)))))))
             (not (exist-LSRM-in-ROB-p idx (MT-step MT MA sigs))))
  :hints (("Goal" :in-theory (enable STEP-SREG-REF lift-b-ops
                                     srname-p rname-p)
                  :cases ((b1p (flush-all? MA sigs))))
          (use-hint-always (:cases ((b1p (dispatch-inst? MA)))))
          (use-hint-always
           (:cases ((b1p (reg-ref-wait?
                          (sreg-tbl-nth (rob-write-rid (MA-rob MA))
                                        (DQ-sreg-tbl (MA-DQ MA))))))))))

;;;; Proof of LSRM-in-ROB-MT-step-if-reg-ref-wait
(encapsulate nil
(local
(defthm LSRM-in-ROB-if-dispatch-inst-help
    (implies (and (inv MT MA)
                  (b1p (dispatch-inst? MA))
                  (b1p (cntlv-wb? (dispatch-cntlv MA)))
                  (b1p (cntlv-wb-sreg? (dispatch-cntlv MA)))
                  (not (b1p (INST-specultv? (INST-at-stg '(DQ 0) MT))))
                  (not (b1p (INST-modified? (INST-at-stg '(DQ 0) MT))))
                  (not (INST-fetch-error-detected-p (INST-at-stg '(DQ 0) MT)))
                  (MAETT-p MT) (MA-state-p MA))
             (sreg-modifier-p (dispatch-dest-reg MA)
                              (INST-at-stg '(DQ 0) MT)))
  :hints (("Goal" :in-theory (enable sreg-modifier-p DQ-OUT-DEST-REG
                                     dispatch-cntlv INST-WB? INST-wb-sreg?
                                     INST-dest-reg INST-cntlv INST-opcode
                                     lift-b-ops decode rdb logbit*
                                     dispatch-dest-reg)))))

; The dispatched instruction is a special register modifier
; if the wb? and wb-sreg? field of the dispatch-cntlv are 1.
(defthm LSRM-in-ROB-if-dispatch-inst
    (implies (and (inv MT MA)
                  (b1p (dispatch-inst? MA))
                  (b1p (cntlv-wb? (dispatch-cntlv MA)))
                  (b1p (cntlv-wb-sreg? (dispatch-cntlv MA)))
                  (not (b1p (flush-all? MA sigs)))
```

```
                (not (b1p (MT-specultv-at-dispatch? (MT-step MT MA sigs))))
                (not (b1p (MT-modified-at-dispatch? (MT-step MT MA sigs))))
                (not (MT-CMI-p (MT-step MT MA sigs)))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (sreg-modifier-p (dispatch-dest-reg MA)
                            (INST-at-stg '(DQ 0) MT))))
)

; The last special register modifier does not commit in this cycle,
; if it is not at the head of the ROB.
(defthm not-INST-commit-LSRM-in-ROB-if-not-ROB-head
    (implies (and (inv MT MA)
                  (not (equal (MT-ROB-head MT)
                              (INST-tag (LSRM-in-ROB idx MT))))
                  (exist-LSRM-in-ROB-p idx MT)
                  (MAETT-p MT) (MA-state-p MA) (srname-p idx))
             (equal (INST-commit? (LSRM-in-ROB idx MT) MA) 0))
  :hints (("Goal" :in-theory (enable INST-commit? equal-b1p-converter
                                     lift-b-ops))))

(encapsulate nil
(local
 (defthm ROBE-dest-INST-dest-reg-special
    (implies (and (inv MT MA)
                  (equal (MT-ROB-head MT) (INST-tag i))
                  (INST-writeback-p i)
                  (complete-stg-p (INST-stg i))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (INST-specultv? i)))
                  (not (INST-fetch-error-detected-p i))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i))
             (equal (robe-dest (nth-robe (MT-ROB-head MT) (MA-rob MA)))
                    (INST-dest-reg i)))))

(local
(defthm not-INST-commit-LSRM-in-ROB-local
    (implies (and (inv MT MA)
                  (not (equal (rob-write-rid (MA-ROB MA)) idx))
                  (exist-LSRM-in-ROB-p idx MT)
                  (not (b1p (INST-specultv? (LSRM-in-ROB idx MT))))
                  (not (b1p (INST-modified? (LSRM-in-ROB idx MT))))
                  (not (b1p (INST-excpt? (LSRM-in-ROB idx MT))))
                  (MAETT-p MT) (MA-state-p MA) (srname-p idx))
             (equal (INST-commit? (LSRM-in-ROB idx MT) MA) 0))
  :hints (("Goal" :in-theory (enable rob-write-rid INST-commit? lift-b-ops
                                     equal-b1p-converter INST-excpt?)))))

; The last special register modifier of register idx does not commit,
; when ROB-write-rid designates another register than idx.
(defthm not-INST-commit-LSRM-in-ROB-if-rob-write-rid-differs
    (implies (and (inv MT MA)
                  (not (equal (rob-write-rid (MA-ROB MA)) idx))
                  (exist-LSRM-in-ROB-p idx MT)
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (MT-specultv-at-dispatch? (MT-step MT MA sigs))))
                  (not (b1p (MT-modified-at-dispatch? (MT-step MT MA sigs))))
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (MAETT-p MT) (MA-state-p MA) (srname-p idx))
             (equal (INST-commit? (LSRM-in-ROB idx MT) MA) 0))
  :hints (("Goal" :in-theory (enable equal-b1p-converter
                                     flush-all? lift-b-ops)
```

```
                  :restrict ((MT-CMI-P-IF-MODIFIED-INST-COMMIT
                              ((i (LSRM-in-ROB idx MT)))))
                  :cases ((b1p (INST-specultv?
                                (LSRM-in-ROB idx MT)))
                          (b1p (INST-modified?
                                (LSRM-in-ROB idx MT)))))
          ("subgoal 3" :cases ((b1p (INST-excpt?
                                     (LSRM-in-ROB idx MT)))))
          ("subgoal 3.1" :cases
                 ((complete-stg-p (INST-stg (LSRM-IN-ROB IDX MT)))))))
)

(encapsulate nil
(local
(defthm ROBE-WB-INST-WB-special
    (implies (and (inv MT MA)
                  (equal (MT-rob-head MT) (INST-tag i))
                  (INST-in i MT) (INST-p i)
                  (not (INST-fetch-error-detected-p i))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (INST-specultv? i)))
                  (dispatched-p i) (not (committed-p i))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (robe-wb? (nth-robe (MT-rob-head MT) (MA-rob MA)))
                    (INST-wb? i)))))

(local
(defthm ROBE-WB-INST-WB-SREG-special
    (implies (and (inv MT MA)
                  (equal (MT-rob-head MT) (INST-tag i))
                  (INST-in i MT) (INST-p i)
                  (not (INST-fetch-error-detected-p i))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (INST-specultv? i)))
                  (dispatched-p i) (not (committed-p i))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (robe-wb-sreg? (nth-robe (MT-rob-head MT) (MA-rob MA)))
                    (INST-wb-sreg? i)))))

(local
 (defthm ROBE-EXCPT-INST-EXCPT-FLAGS-special
    (implies (and (inv MT MA)
                  (equal (MT-ROB-head MT) (INST-tag i))
                  (complete-stg-p (INST-stg i))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (INST-specultv? i)))
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i))
             (equal (robe-excpt (nth-robe (MT-ROB-head MT) (MA-rob MA)))
                    (INST-excpt-flags i)))))

(local
(defthm not-INST-commit-LSRM-in-ROB-help
    (implies (and (inv MT MA)
                  (not (b1p (ROB-write-sreg? (MA-ROB MA))))
                  (not (b1p (INST-specultv? (LSRM-in-ROB idx MT))))
                  (not (b1p (INST-modified? (LSRM-in-ROB idx MT))))
                  (not (b1p (INST-excpt? (LSRM-in-ROB idx MT))))
                  (exist-LSRM-in-ROB-p idx MT)
                  (MAETT-p MT) (MA-state-p MA) (srname-p idx))
             (equal (INST-commit? (LSRM-in-ROB idx MT) MA) 0))
  :hints (("Goal" :in-theory
```

```
                (e/d (INST-commit? lift-b-ops
                      INST-excpt? equal-b1p-converter
                      COMMIT-INST? ROB-write-sreg?)
                     (uniq-inst-of-tag-if-commit-inst
                      COMPLETE-INST-OF-TAG-IF-COMMIT-INST
                      NOT-EXECUTE-STG-P-INST-AT-ROB-HEAD-IF-COMMIT-INST))))))

; If ROB-write-sreg is 0, a special register modifier does not commit.
(defthm not-INST-commit-LSRM-in-ROB
    (implies (and (inv MT MA)
                  (not (b1p (ROB-write-sreg? (MA-ROB MA))))
                  (NOT (B1P (MT-SPECULTV-AT-DISPATCH? (MT-STEP MT MA SIGS))))
                  (NOT (B1P (MT-MODIFIED-AT-DISPATCH? (MT-STEP MT MA SIGS))))
                  (NOT (MT-CMI-P (MT-STEP MT MA SIGS)))
                  (NOT (B1P (FLUSH-ALL? MA SIGS)))
                  (exist-LSRM-in-ROB-p idx MT)
                  (MAETT-p MT) (MA-state-p MA) (srname-p idx))
             (equal (INST-commit? (LSRM-in-ROB idx MT) MA) 0))
  :hints (("Goal" :in-theory (enable equal-b1p-converter
                                     flush-all? lift-b-ops)
                  :restrict ((MT-CMI-P-IF-MODIFIED-INST-COMMIT
                              ((i (LSRM-in-ROB idx MT)))))
                  :cases ((b1p (INST-specultv?
                                (LSRM-in-ROB idx MT)))
                          (b1p (INST-modified?
                                (LSRM-in-ROB idx MT)))))
          ("subgoal 3" :cases ((b1p (INST-excpt?
                                     (LSRM-in-ROB idx MT)))))
          ("subgoal 3.1" :cases
                 ((complete-stg-p (INST-stg (LSRM-IN-ROB IDX MT)))))))
)

(encapsulate nil
(local
(defthm LSRM-in-ROB-MT-step-if-inst-commit-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (trace-exist-LSRM-in-ROB-p idx trace)
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (inst-commit?
                             (trace-LSRM-in-ROB idx trace)
                             MA)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (trace-exist-LSRM-in-ROB-p idx
                    (step-trace trace MT MA sigs ISA spc smc)))))

; If there are special register modifiers in the ROB, and no instruction
; commits in this cycle, then the ROB continues to have the modifiers
; in the next cycle.
(defthm LSRM-in-ROB-MT-step-if-inst-commit
    (implies (and (inv MT MA)
                  (exist-LSRM-in-ROB-p idx MT)
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (inst-commit? (LSRM-in-ROB idx MT) MA)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (exist-LSRM-in-ROB-p idx (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable exist-LSRM-in-ROB-p
                                     LSRM-in-ROB))))
)

(encapsulate nil
(local
```

```
(defthm LRM-in-ROB-MT-step-if-dispatch-induct
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (member-equal i trace) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (dispatch-inst? MA))
                  (equal (INST-stg i) '(DQ 0))
                  (sreg-modifier-p idx i)
                  (MAETT-p MT) (MA-state-p MA))
             (trace-exist-LSRM-in-ROB-p idx
                       (step-trace trace MT MA sigs ISA spc smc)))))

(local
(defthm LRM-in-ROB-MT-step-if-dispatch-help
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (dispatch-inst? MA))
                  (equal (INST-stg i) '(DQ 0))
                  (sreg-modifier-p idx i)
                  (MAETT-p MT) (MA-state-p MA))
             (exist-LSRM-in-ROB-p idx (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable exist-LSRM-in-ROB-p INST-in)))))

; If a special register modifier is dispatched, then there will
; be modifiers in the ROB.
(defthm LSRM-in-ROB-MT-step-if-dispatch
    (implies (and (inv MT MA)
                  (b1p (dispatch-inst? MA))
                  (not (b1p (flush-all? MA sigs)))
                  (sreg-modifier-p idx (INST-at-stg '(DQ 0) MT))
                  (MAETT-p MT) (MA-state-p MA))
             (exist-LSRM-in-ROB-p idx (MT-step MT MA sigs)))
  :hints (("Goal" :restrict ((LRM-in-ROB-MT-step-if-dispatch-help
                              ((i (INST-at-stg '(DQ 0) MT))))))))
)

;  A landmark lemma
;  There will be special register modifiers in the ROB in the next cycle
;  if the wait? bit of the special register reference table will be 1.
(defthm LSRM-in-ROB-MT-step-if-reg-ref-wait
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (srname-p idx)
                  (not (b1p (MT-specultv-at-dispatch? (MT-step MT MA sigs))))
                  (not (b1p (MT-modified-at-dispatch? (MT-step MT MA sigs))))
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (b1p (reg-ref-wait?
                        (sreg-tbl-nth idx
                                      (DQ-sreg-tbl (step-DQ MA sigs))))))
             (exist-LSRM-in-ROB-p idx (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable STEP-SREG-REF lift-b-ops srname-p
                                     rname-p))))

;;;;;;; Proof of INST-tag-LSRM-MT-step
(encapsulate nil
(local
(defthm trace-LSRM-in-ROB-step-trace-if-dispatch-inst
    (implies (and (inv MT MA)
                  (member-equal i trace) (INST-p i)
                  (subtrace-p trace MT) (INST-listp trace)
                  (not (b1p (flush-all? MA sigs)))
```

```
                (b1p (dispatch-inst? MA))
                (equal (INST-stg i) '(DQ 0))
                (sreg-modifier-p idx i)
                (MAETT-p MT) (MA-state-p MA))
           (trace-exist-LSRM-in-ROB-p idx
                       (step-trace trace MT MA sigs ISA spc smc)))))

(local
(defthm LSRM-in-ROB-MT-step-if-dispatch-inst-induct
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (sreg-modifier-p idx i)
                (member-equal i trace) (INST-p i)
                (not (b1p (flush-all? MA sigs)))
                (b1p (dispatch-inst? MA))
                (equal (INST-stg i) '(DQ 0))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (equal (trace-LSRM-in-ROB idx
                       (step-trace trace MT MA sigs ISA spc smc))
             (step-INST i MT MA sigs)))
  :hints (("goal" :restrict
                ((NOT-TRACE-LSRM-IN-ROB-STEP-TRACE
                  ((i (car trace)))))))))

(local
(defthm LSRM-in-ROB-MT-step-if-dispatch-inst-help
    (implies (and (inv MT MA)
                (sreg-modifier-p idx i)
                (INST-in i MT) (INST-p i)
                (not (b1p (flush-all? MA sigs)))
                (b1p (dispatch-inst? MA))
                (equal (INST-stg i) '(DQ 0))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (equal (LSRM-in-ROB idx (MT-step MT MA sigs))
                  (step-INST i MT MA sigs)))
  :hints (("Goal" :in-theory (enable LSRM-in-ROB
                                     INST-in)))))

; The dispatched instruction is the last special register modifier
; if the dispatched instruction is a modifier.
(defthm LSRM-in-ROB-MT-step-if-dispatch-inst
    (implies (and (inv MT MA)
                (b1p (dispatch-inst? MA))
                (not (b1p (flush-all? MA sigs)))
                (sreg-modifier-p idx (INST-at-stg '(DQ 0) MT))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (equal (LSRM-in-ROB idx (MT-step MT MA sigs))
                  (step-INST (INST-at-stg '(DQ 0) MT) MT MA sigs)))
  :hints (("Goal" :restrict
                ((LSRM-in-ROB-MT-step-if-dispatch-inst-help
                  ((i (INST-at-stg '(DQ 0) MT)))))))))
)

(local
(defthm trace-LSRM-in-ROB-step-trace-if-not-INST-commit
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (trace-exist-LSRM-in-ROB-p idx trace)
                (not (b1p (flush-all? MA sigs)))
                (not (b1p (INST-commit?
                           (trace-LSRM-in-ROB idx trace) MA)))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
```

```
            (trace-exist-LSRM-in-ROB-p idx
               (step-trace trace MT MA sigs ISA spc smc)))))

(encapsulate nil
(local
(defthm LSRM-in-ROB-MT-step-if-not-dispatch-inst-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (not (b1p (dispatch-inst? MA)))
                  (not (b1p (flush-all? MA sigs)))
                  (trace-exist-LSRM-in-ROB-p idx trace)
                  (not (b1p (INST-commit?
                             (trace-LSRM-in-ROB idx trace) MA)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (srname-p idx))
             (equal (trace-LSRM-in-ROB idx
                           (step-trace trace MT MA sigs ISA spc smc))
               (step-INST (trace-LSRM-in-ROB idx trace)
                          MT MA sigs)))))

; The last special register modifier in the current cycle is also the
; last register modifier in the next cycle, if no instruction is
; dispatched during this cycle.
(defthm LSRM-in-ROB-MT-step-if-not-dispatch-inst
    (implies (and (inv MT MA)
                  (not (b1p (dispatch-inst? MA)))
                  (not (b1p (flush-all? MA sigs)))
                  (exist-LSRM-in-ROB-p idx MT)
                  (not (b1p (INST-commit? (LSRM-in-ROB idx MT) MA)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (srname-p idx))
             (equal (LSRM-in-ROB idx (MT-step MT MA sigs))
                    (step-INST (LSRM-in-ROB idx MT) MT MA sigs)))
  :hints (("Goal" :in-theory (enable LSRM-in-ROB
                                     exist-LSRM-in-ROB-p))))

)

(encapsulate nil
(local
(defthm LSRM-in-ROB-MT-step-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (not (b1p (flush-all? MA sigs)))
                  (not (sreg-modifier-p idx
                                        (INST-at-stg-in-trace '(DQ 0) trace)))
                  (trace-exist-LSRM-in-ROB-p idx trace)
                  (not (b1p (INST-commit?
                             (trace-LSRM-in-ROB idx trace) MA)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (srname-p idx))
             (equal (trace-LSRM-in-ROB idx
                        (step-trace trace MT MA sigs ISA spc smc))
               (step-INST (trace-LSRM-in-ROB idx trace)
                          MT MA sigs)))))

; A landmark lemma
; The last special register modifier continues to be the last modifier,
; if the instruction at DQ0 is not a modifier.
(defthm LSRM-in-ROB-MT-step
    (implies (and (inv MT MA)
                  (not (b1p (flush-all? MA sigs)))
```

```
                (not (sreg-modifier-p idx (INST-at-stg '(DQ 0) MT)))
                (exist-LSRM-in-ROB-p idx MT)
                (not (b1p (INST-commit? (LSRM-in-ROB idx MT) MA)))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                (srname-p idx))
           (equal (LSRM-in-ROB idx (MT-step MT MA sigs))
                  (step-INST (LSRM-in-ROB idx MT) MT MA sigs)))
  :hints (("Goal" :in-theory (enable LSRM-in-ROB
                                     INST-at-stg
                                     exist-LSRM-in-ROB-p))))
)

; The proof of INST-tag-LSRM-MT-step is divided into three
; cases, where dispatch-inst? is 0, where the instruction at DQ0 is not a
; register modifier, and where both conditions are 1.
(defthm INST-tag-LSRM-MT-step-if-not-dispatch-inst
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                (srname-p idx)
                (not (b1p (dispatch-inst? MA)))
                (not (b1p (MT-specultv-at-dispatch? (MT-step MT MA sigs))))
                (not (b1p (MT-modified-at-dispatch? (MT-step MT MA sigs))))
                (not (MT-CMI-p (MT-step MT MA sigs)))
                (b1p (reg-ref-wait?
                      (step-sreg-ref (sreg-tbl-nth idx
                                                   (DQ-sreg-tbl (MA-DQ MA)))
                                     idx MA sigs))))
           (equal (INST-tag (LSRM-in-ROB
                             idx (MT-step MT MA sigs)))
                  (reg-ref-tag (step-sreg-ref
                                (sreg-tbl-nth idx (DQ-sreg-tbl (MA-DQ MA)))
                                idx MA sigs))))
  :hints (("Goal" :in-theory (enable step-sreg-ref lift-b-ops
                                     rname-p srname-p))))

(defthm INST-tag-LSRM-MT-step-if-not-sreg-modifier
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                (srname-p idx)
                (not (sreg-modifier-p idx (INST-at-stg '(DQ 0) MT)))
                (not (b1p (MT-specultv-at-dispatch? (MT-step MT MA sigs))))
                (not (b1p (MT-modified-at-dispatch? (MT-step MT MA sigs))))
                (not (MT-CMI-p (MT-step MT MA sigs)))
                (b1p (reg-ref-wait?
                      (step-sreg-ref (sreg-tbl-nth idx
                                                   (DQ-sreg-tbl (MA-DQ MA)))
                                     idx MA sigs))))
           (equal (INST-tag (LSRM-in-ROB
                             idx (MT-step MT MA sigs)))
                  (reg-ref-tag (step-sreg-ref
                                (sreg-tbl-nth idx (DQ-sreg-tbl (MA-DQ MA)))
                                idx MA sigs))))
  :hints (("Goal" :in-theory (enable step-sreg-ref lift-b-ops
                                     srname-p))))

(defthm INST-tag-LSRM-MT-step-if-dispatch-inst
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                (srname-p idx)
                (b1p (dispatch-inst? MA))
                (sreg-modifier-p idx (INST-at-stg '(DQ 0) MT))
                (not (b1p (MT-specultv-at-dispatch? (MT-step MT MA sigs))))
```

```
                (not (b1p (MT-modified-at-dispatch? (MT-step MT MA sigs))))
                (not (MT-CMI-p (MT-step MT MA sigs)))
                (b1p (reg-ref-wait?
                      (step-sreg-ref (sreg-tbl-nth idx
                                                   (DQ-sreg-tbl (MA-DQ MA)))
                                     idx MA sigs))))
           (equal (reg-ref-tag (step-sreg-ref
                                (sreg-tbl-nth idx (DQ-sreg-tbl (MA-DQ MA)))
                                idx MA sigs))
                  (INST-tag (LSRM-in-ROB
                             idx (MT-step MT MA sigs)))))
  :hints (("Goal" :in-theory (enable step-sreg-ref lift-b-ops srname-p))))

; An important lemma. Summarizing the three lemmas above.
; The new value of tag field of register reference table is the Tomasulo's
; tag of the last special register modifier.
(defthm INST-tag-LSRM-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (srname-p idx)
                  (not (b1p (MT-specultv-at-dispatch? (MT-step MT MA sigs))))
                  (not (b1p (MT-modified-at-dispatch? (MT-step MT MA sigs))))
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (b1p (reg-ref-wait?
                        (step-sreg-ref (sreg-tbl-nth idx
                                                     (DQ-sreg-tbl (MA-DQ MA)))
                                       idx MA sigs))))
             (equal (INST-tag (LSRM-in-ROB
                               idx (MT-step MT MA sigs)))
                    (reg-ref-tag (step-sreg-ref
                                  (sreg-tbl-nth idx (DQ-sreg-tbl (MA-DQ MA)))
                                  idx MA sigs))))
  :hints (("Goal" :cases
                  ((not (b1p (dispatch-inst? MA)))
                   (not (sreg-modifier-p idx (INST-at-stg '(DQ 0) MT)))))))

; A landmark lemma.
; consistent-sreg-ref-p is true for the register idx.
(defthm consistent-sreg-ref-p-MT-step
    (implies (and (inv MT MA)
                  (srname-p idx)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (not (MT-CMI-p (MT-step MT MA sigs))))
             (consistent-sreg-ref-p idx (MT-step MT MA sigs)
                                    (MA-step MA sigs)))
  :hints (("Goal" :in-theory (enable consistent-sreg-ref-p))))

; A landmark lemma.  Consistent-sreg-tbl-p is preserved.
(defthm consistent-sreg-tbl-p-preserved
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (not (MT-CMI-p (MT-step MT MA sigs))))
             (consistent-sreg-tbl-p (MT-step MT MA sigs)
                                    (MA-step MA sigs)))
  :hints (("Goal" :in-theory (enable consistent-sreg-tbl-p))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Proof of consistent-RS-p
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Rules about Common Data Bus and instruction stage.
; If the common data bus is not available, the instruction cannot make
```

```
; progress.
; If instruction i is currently at the integer unit, and the Common data
; bus is not ready for instruction i, then i will stay in the integer unit.
; Similar lemmas are proven for MU, BU, and LSU in
;  MU-stg-p-step-inst-if-not-CDB-ready-for
;  BU-stg-p-step-inst-if-not-CDB-ready-for
;  LSU-stg-p-step-inst-if-not-CDB-ready-for
(defthm IU-stg-p-step-inst-if-not-CDB-ready-for
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (IU-stg-p (INST-stg i))
                  (not (b1p (CDB-ready-for? (INST-tag i) MA))))
             (IU-stg-p (INST-stg (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (enable IU-stg-p
                                     INST-STG-STEP-INST-IU-RS0
                                     INST-STG-STEP-INST-IU-RS1
                                     lift-b-ops CDB-tag
                                     IU-OUTPUT-tag
                                     CDB-ready-for? CDB-ready?
                                     IU-RS1-ISSUE-READY?
                                     CDB-FOR-IU?
                                     issue-IU-RS0? issue-IU-RS1?))))

(defthm MU-stg-p-step-inst-if-not-CDB-ready-for
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (MU-stg-p (INST-stg i))
                  (not (b1p (CDB-ready-for? (INST-tag i) MA))))
             (MU-stg-p (INST-stg (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (enable MU-stg-p
                                     CDB-FOR-IU?
                                     INST-STG-STEP-INST-MU-RS0
                                     INST-STG-STEP-INST-MU-RS1
                                     INST-STG-STEP-INST-MU-LCH1
                                     INST-STG-STEP-INST-MU-LCH2
                                     lift-b-ops CDB-tag
                                     CDB-FOR-MU? CDB-ready-for? CDB-ready?
                                     IU-output-tag CDB-FOR-BU?))))


(defthm BU-stg-p-step-inst-if-not-CDB-ready-for
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (BU-stg-p (INST-stg i))
                  (not (b1p (CDB-ready-for? (INST-tag i) MA))))
             (BU-stg-p (INST-stg (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (enable BU-stg-p CDB-ready-for?
                                     CDB-READY? CDB-for-BU?
                                     INST-STG-STEP-INST-BU-RS0
                                     INST-STG-STEP-INST-BU-RS1
                                     lift-b-ops CDB-tag
                                     CDB-FOR-IU? ISSUE-BU-RS0?
                                     ISSUE-BU-RS1?
                                     BU-RS1-ISSUE-READY?
                                     BU-OUTPUT-tag))))

(defthm LSU-stg-p-step-inst-if-not-CDB-ready-for
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
```

```
                (MAETT-p MT) (MA-state-p MA)
                (LSU-stg-p (INST-stg i))
                (not (b1p (CDB-ready-for? (INST-tag i) MA))))
           (LSU-stg-p (INST-stg (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (enable LSU-stg-p CDB-ready-for? lift-b-ops
                                     CDB-READY? CDB-for-LSU?
                                     INST-STG-STEP-INST-LSU-RS0
                                     INST-STG-STEP-INST-LSU-RS1
                                     INST-STG-STEP-INST-LSU-RBUF
                                     INST-STG-STEP-INST-LSU-WBUF1-LCH
                                     INST-STG-STEP-INST-LSU-WBUF0
                                     INST-STG-STEP-INST-LSU-WBUF1
                                     INST-stg-step-INST-LSU-wbuf0-lch
                                     CDB-tag))))

; Summarize the four lemmas above.  If the instruction i is in the
; execution stage, and the common data bus is not ready for it, i will stay
; in execution stage.
(defthm execute-stg-p-step-inst-if-not-CDB-ready-for
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA)
                (execute-stg-p (INST-stg i))
                (not (b1p (CDB-ready-for? (INST-tag i) MA))))
           (execute-stg-p (INST-stg (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (enable execute-stg-p))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Rewriting Rules derived from consistent-RS-p
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
(local
(defthm snrname-p-INST-ra-if-not-decode-error
    (implies (and (INST-p i)
                (not (b1p (INST-decode-error? i)))
                (b1p (logbit 3 (cntlv-operand (INST-cntlv i)))))
           (srname-p (INST-ra i)))
  :hints (("goal" :in-theory (enable srname-p INST-decode-error?
                                     decode-illegal-inst? INST-opcode
                                     decode rdb logbit*
                                     INST-cntlv lift-b-ops)))))

; Some help lemmas about cntlv
(defthm cntlv-operand0-if-logbit1-exunit
    (implies (and (INST-p i) (b1p (logbit 1 (cntlv-exunit (INST-cntlv i)))))
           (equal (logbit 0 (cntlv-operand (INST-cntlv i))) 1))
  :hints (("Goal" :in-theory (enable INST-cntlv lift-b-ops
                                     equal-b1p-converter
                                     decode logbit* rdb))))

(defthm cntlv-operand0-if-logbit2-exunit
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(DQ 0))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (INST-fetch-error-detected-p i))
                (b1p (logbit 2 (cntlv-exunit (INST-cntlv i))))
                (not (b1p (dispatch-ready1? MA)))
                (MAETT-p MT) (MA-state-p MA))
           (equal (logbit 0 (cntlv-operand (INST-cntlv i))) 1))
  :hints (("Goal" :in-theory (enable INST-cntlv lift-b-ops
```

```
                                    dispatch-ready1? DQ-out-ready1?
                                    equal-b1p-converter
                                    decode logbit* rdb))))

(defthm cntlv-operand0-if-logbit3-exunit
    (implies (and (INST-p i) (b1p (logbit 3 (cntlv-exunit (INST-cntlv i)))))
             (equal (logbit 2 (cntlv-operand (INST-cntlv i))) 1))
  :hints (("Goal" :in-theory (enable INST-cntlv lift-b-ops
                                    equal-b1p-converter
                                    decode logbit* rdb))))

; Lemmas derived from consistent-RS-p
; These rules are used to prove the invariantness lemma for consistent-RS-p.
(encapsulate nil
(local
(defthm consistent-RS-entry-p-inst-in-help
    (implies (and (trace-consistent-RS-p trace MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (member-equal i trace) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA))
             (consistent-RS-entry-p i MT MA))))

; If invariants hold for MT and MA, consistent-RS-entry-p is
; correct for any instruction in MT.  Intuitively, the reservation
; station storing i contains correct values.
(defthm consistent-RS-entry-p-inst-in
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA))
             (consistent-RS-entry-p i MT MA))
  :hints (("Goal" :in-theory (enable inv consistent-RS-p
                                    INST-in)))
  :rule-classes nil)
)
;
; Following 61 lemmas are direct consequence from consistent-RS-entry-p.
;
; Please see the comments on Consistent-RS-p in invariants-def.lisp for
; their implication.  We convert the invariants consistent-RS-p into
; rewriting rules.
;
; There are some possibilities of generating these lemmas automatically,
; but we have not tried yet.
;
(defthm exist-LRM-before-p-IU-RS0-if-logbit0-ra
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(IU RS0))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (RS-ready1? (IU-RS0 (MA-IU MA)))))
                  (b1p (logbit 0 (cntlv-operand (INST-cntlv i))))
                  (MAETT-p MT) (MA-state-p MA))
             (exist-LRM-before-p i (INST-ra i) MT))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                    CONSISTENT-IU-RS-P
                                    CONSISTENT-IU-RS0-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm execute-stg-p-LRM-before-IU-RS0-if-logbit0-ra
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
```

```
                (equal (INST-stg i) '(IU RS0))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready1? (IU-RS0 (MA-IU MA)))))
                (b1p (logbit 0 (cntlv-operand (INST-cntlv i))))
                (MAETT-p MT) (MA-state-p MA))
           (execute-stg-p (INST-stg (LRM-before i (INST-ra i) MT))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-IU-RS-P
                                     CONSISTENT-IU-RS0-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm INST-tag-LRM-before-IU-RS0-if-logbit0-ra
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(IU RS0))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready1? (IU-RS0 (MA-IU MA)))))
                (b1p (logbit 0 (cntlv-operand (INST-cntlv i))))
                (MAETT-p MT) (MA-state-p MA))
           (equal (INST-tag (LRM-before i (INST-ra i) MT))
                  (RS-src1 (IU-RS0 (MA-IU MA)))))
    :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                       CONSISTENT-IU-RS-P
                                       CONSISTENT-IU-RS0-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm exist-LRM-before-p-IU-RS0-if-logbit0-rb
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(IU RS0))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready2? (IU-RS0 (MA-IU MA)))))
                (b1p (logbit 0 (cntlv-operand (INST-cntlv i))))
                (MAETT-p MT) (MA-state-p MA))
           (exist-LRM-before-p i (INST-rb i) MT))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-IU-RS-P
                                     CONSISTENT-IU-RS0-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm execute-stg-p-LRM-before-IU-RS0-if-logbit0-rb
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(IU RS0))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready2? (IU-RS0 (MA-IU MA)))))
                (b1p (logbit 0 (cntlv-operand (INST-cntlv i))))
                (MAETT-p MT) (MA-state-p MA))
           (execute-stg-p (INST-stg (LRM-before i (INST-rb i) MT))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-IU-RS-P
                                     CONSISTENT-IU-RS0-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm INST-tag-LRM-before-IU-RS0-if-logbit0-rb
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(IU RS0))
```

```
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready2? (IU-RS0 (MA-IU MA)))))
                (b1p (logbit 0 (cntlv-operand (INST-cntlv i))))
                (MAETT-p MT) (MA-state-p MA))
           (equal (INST-tag (LRM-before i (INST-rb i) MT))
                  (RS-src2 (IU-RS0 (MA-IU MA)))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-IU-RS-P
                                     CONSISTENT-IU-RS0-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm exist-LRM-before-p-IU-RS0-if-logbit2
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(IU RS0))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready1? (IU-RS0 (MA-IU MA)))))
                (b1p (logbit 2 (cntlv-operand (INST-cntlv i))))
                (MAETT-p MT) (MA-state-p MA))
           (exist-LRM-before-p i (INST-rc i) MT))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-IU-RS-P
                                     CONSISTENT-IU-RS0-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm execute-stg-p-LRM-before-IU-RS0-if-logbit2
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(IU RS0))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready1? (IU-RS0 (MA-IU MA)))))
                (b1p (logbit 2 (cntlv-operand (INST-cntlv i))))
                (MAETT-p MT) (MA-state-p MA))
           (execute-stg-p (INST-stg (LRM-before i (INST-rc i) MT))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-IU-RS-P
                                     CONSISTENT-IU-RS0-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm INST-tag-LRM-before-IU-RS0-if-logbit2
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(IU RS0))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready1? (IU-RS0 (MA-IU MA)))))
                (b1p (logbit 2 (cntlv-operand (INST-cntlv i))))
                (MAETT-p MT) (MA-state-p MA))
           (equal (INST-tag (LRM-before i (INST-rc i) MT))
                  (RS-src1 (IU-RS0 (MA-IU MA)))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-IU-RS-P
                                     CONSISTENT-IU-RS0-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm exist-LSRM-before-p-IU-RS0-if-logbit3
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(IU RS0))
```

```
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready1? (IU-RS0 (MA-IU MA)))))
                (b1p (logbit 3 (cntlv-operand (INST-cntlv i))))
                (MAETT-p MT) (MA-state-p MA))
           (exist-LSRM-before-p i (INST-ra i) MT))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-IU-RS-P
                                     CONSISTENT-IU-RS0-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm execute-stg-p-LSRM-before-IU-RS0-if-logbit3
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(IU RS0))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready1? (IU-RS0 (MA-IU MA)))))
                (b1p (logbit 3 (cntlv-operand (INST-cntlv i))))
                (MAETT-p MT) (MA-state-p MA))
           (execute-stg-p (INST-stg (LSRM-before i (INST-ra i) MT))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-IU-RS-P
                                     CONSISTENT-IU-RS0-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm INST-tag-LSRM-before-IU-RS0-if-logbit3
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(IU RS0))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready1? (IU-RS0 (MA-IU MA)))))
                (b1p (logbit 3 (cntlv-operand (INST-cntlv i))))
                (MAETT-p MT) (MA-state-p MA))
           (equal (INST-tag (LSRM-before i (INST-ra i) MT))
                  (RS-src1 (IU-RS0 (MA-IU MA)))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-IU-RS-P
                                     CONSISTENT-IU-RS0-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm exist-LRM-before-p-IU-RS1-if-logbit0-ra
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(IU RS1))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready1? (IU-RS1 (MA-IU MA)))))
                (b1p (logbit 0 (cntlv-operand (INST-cntlv i))))
                (MAETT-p MT) (MA-state-p MA))
           (exist-LRM-before-p i (INST-ra i) MT))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-IU-RS-P
                                     CONSISTENT-IU-RS1-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm execute-stg-p-LRM-before-IU-RS1-if-logbit0-ra
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(IU RS1))
                (not (b1p (INST-specultv? i)))
```

```
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready1? (IU-RS1 (MA-IU MA)))))
                (b1p (logbit 0 (cntlv-operand (INST-cntlv i))))
                (MAETT-p MT) (MA-state-p MA))
           (execute-stg-p (INST-stg (LRM-before i (INST-ra i) MT))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-IU-RS-P
                                     CONSISTENT-IU-RS1-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm INST-tag-LRM-before-if-IU-RS1-logbit0-ra
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(IU RS1))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready1? (IU-RS1 (MA-IU MA)))))
                (b1p (logbit 0 (cntlv-operand (INST-cntlv i))))
                (MAETT-p MT) (MA-state-p MA))
           (equal (INST-tag (LRM-before i (INST-ra i) MT))
                  (RS-src1 (IU-RS1 (MA-IU MA)))))
    :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                       CONSISTENT-IU-RS-P
                                       CONSISTENT-IU-RS1-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm exist-LRM-before-p-if-IU-RS1-logbit0-rb
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(IU RS1))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready2? (IU-RS1 (MA-IU MA)))))
                (b1p (logbit 0 (cntlv-operand (INST-cntlv i))))
                (MAETT-p MT) (MA-state-p MA))
           (exist-LRM-before-p i (INST-rb i) MT))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-IU-RS-P
                                     CONSISTENT-IU-RS1-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm execute-stg-p-LRM-before-if-IU-RS1-logbit0-rb
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(IU RS1))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready2? (IU-RS1 (MA-IU MA)))))
                (b1p (logbit 0 (cntlv-operand (INST-cntlv i))))
                (MAETT-p MT) (MA-state-p MA))
           (execute-stg-p (INST-stg (LRM-before i (INST-rb i) MT))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-IU-RS-P
                                     CONSISTENT-IU-RS1-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm INST-tag-LRM-before-if-IU-RS1-logbit0-rb
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(IU RS1))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
```

```
                (not (b1p (RS-ready2? (IU-RS1 (MA-IU MA)))))
                (b1p (logbit 0 (cntlv-operand (INST-cntlv i))))
                (MAETT-p MT) (MA-state-p MA))
           (equal (INST-tag (LRM-before i (INST-rb i) MT))
                  (RS-src2 (IU-RS1 (MA-IU MA)))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-IU-RS-P
                                     CONSISTENT-IU-RS1-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm exist-LRM-before-p-if-IU-RS1-logbit2
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(IU RS1))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready1? (IU-RS1 (MA-IU MA)))))
                (b1p (logbit 2 (cntlv-operand (INST-cntlv i))))
                (MAETT-p MT) (MA-state-p MA))
           (exist-LRM-before-p i (INST-rc i) MT))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-IU-RS-P
                                     CONSISTENT-IU-RS1-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm execute-stg-p-LRM-before-if-IU-RS1-logbit2
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(IU RS1))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready1? (IU-RS1 (MA-IU MA)))))
                (b1p (logbit 2 (cntlv-operand (INST-cntlv i))))
                (MAETT-p MT) (MA-state-p MA))
           (execute-stg-p (INST-stg (LRM-before i (INST-rc i) MT))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-IU-RS-P
                                     CONSISTENT-IU-RS1-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm INST-tag-LRM-before-if-IU-RS1-logbit2
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(IU RS1))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready1? (IU-RS1 (MA-IU MA)))))
                (b1p (logbit 2 (cntlv-operand (INST-cntlv i))))
                (MAETT-p MT) (MA-state-p MA))
           (equal (INST-tag (LRM-before i (INST-rc i) MT))
                  (RS-src1 (IU-RS1 (MA-IU MA)))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-IU-RS-P
                                     CONSISTENT-IU-RS1-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm exist-LSRM-before-p-if-IU-RS1-logbit3
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(IU RS1))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
```

```
                (not (b1p (RS-ready1? (IU-RS1 (MA-IU MA)))))
                (b1p (logbit 3 (cntlv-operand (INST-cntlv i))))
                (MAETT-p MT) (MA-state-p MA))
           (exist-LSRM-before-p i (INST-ra i) MT))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-IU-RS-P
                                     CONSISTENT-IU-RS1-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm execute-stg-p-LSRM-before-if-IU-RS1-logbit3
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(IU RS1))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready1? (IU-RS1 (MA-IU MA)))))
                (b1p (logbit 3 (cntlv-operand (INST-cntlv i))))
                (MAETT-p MT) (MA-state-p MA))
           (execute-stg-p (INST-stg (LSRM-before i (INST-ra i) MT))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-IU-RS-P
                                     CONSISTENT-IU-RS1-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm INST-tag-LSRM-before-if-IU-RS1-logbit3
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(IU RS1))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready1? (IU-RS1 (MA-IU MA)))))
                (b1p (logbit 3 (cntlv-operand (INST-cntlv i))))
                (MAETT-p MT) (MA-state-p MA))
           (equal (INST-tag (LSRM-before i (INST-ra i) MT))
                  (RS-src1 (IU-RS1 (MA-IU MA)))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-IU-RS-P
                                     CONSISTENT-IU-RS1-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm exist-LRM-before-p-if-MU-RS0-ra
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(MU RS0))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready1? (MU-RS0 (MA-MU MA)))))
                (MAETT-p MT) (MA-state-p MA))
           (exist-LRM-before-p i (INST-ra i) MT))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-MU-RS-P
                                     CONSISTENT-MU-RS0-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm execute-stg-p-LRM-before-if-MU-RS0-ra
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(MU RS0))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready1? (MU-RS0 (MA-MU MA)))))
                (MAETT-p MT) (MA-state-p MA))
```

```
            (execute-stg-p (INST-stg (LRM-before i (INST-ra i) MT))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-MU-RS-P
                                     CONSISTENT-MU-RS0-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm INST-tag-LRM-before-if-MU-RS0-ra
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(MU RS0))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (RS-ready1? (MU-RS0 (MA-MU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-tag (LRM-before i (INST-ra i) MT))
                    (RS-src1 (MU-RS0 (MA-MU MA)))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-MU-RS-P
                                     CONSISTENT-MU-RS0-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm exist-LRM-before-p-if-MU-RS0-rb
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(MU RS0))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (RS-ready2? (MU-RS0 (MA-MU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (exist-LRM-before-p i (INST-rb i) MT))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-MU-RS-P
                                     CONSISTENT-MU-RS0-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm execute-stg-p-LRM-before-if-MU-RS0-rb
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(MU RS0))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (RS-ready2? (MU-RS0 (MA-MU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (execute-stg-p (INST-stg (LRM-before i (INST-rb i) MT))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-MU-RS-P
                                     CONSISTENT-MU-RS0-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm INST-tag-LRM-before-if-MU-RS0-rb
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(MU RS0))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (RS-ready2? (MU-RS0 (MA-MU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-tag (LRM-before i (INST-rb i) MT))
                    (RS-src2 (MU-RS0 (MA-MU MA)))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-MU-RS-P
                                     CONSISTENT-MU-RS0-P)
```

```
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm exist-LRM-before-p-if-MU-RS1-ra
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(MU RS1))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (RS-ready1? (MU-RS1 (MA-MU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (exist-LRM-before-p i (INST-ra i) MT))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-MU-RS-P
                                     CONSISTENT-MU-RS1-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm execute-stg-p-LRM-before-if-MU-RS1-ra
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(MU RS1))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (RS-ready1? (MU-RS1 (MA-MU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (execute-stg-p (INST-stg (LRM-before i (INST-ra i) MT))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-MU-RS-P
                                     CONSISTENT-MU-RS1-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm INST-tag-LRM-before-if-MU-RS1-ra
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(MU RS1))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (RS-ready1? (MU-RS1 (MA-MU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-tag (LRM-before i (INST-ra i) MT))
                    (RS-src1 (MU-RS1 (MA-MU MA)))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-MU-RS-P
                                     CONSISTENT-MU-RS1-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm exist-LRM-before-p-if-MU-RS1-rb
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(MU RS1))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (RS-ready2? (MU-RS1 (MA-MU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (exist-LRM-before-p i (INST-rb i) MT))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-MU-RS-P
                                     CONSISTENT-MU-RS1-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm execute-stg-p-LRM-before-if-MU-RS1-rb
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
```

```
                (equal (INST-stg i) '(MU RS1))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready2? (MU-RS1 (MA-MU MA)))))
                (MAETT-p MT) (MA-state-p MA))
           (execute-stg-p (INST-stg (LRM-before i (INST-rb i) MT))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-MU-RS-P
                                     CONSISTENT-MU-RS1-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm INST-tag-LRM-before-if-MU-RS1-rb
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(MU RS1))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (RS-ready2? (MU-RS1 (MA-MU MA)))))
                (MAETT-p MT) (MA-state-p MA))
           (equal (INST-tag (LRM-before i (INST-rb i) MT))
                  (RS-src2 (MU-RS1 (MA-MU MA)))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-MU-RS-P
                                     CONSISTENT-MU-RS1-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm INST-tag-LRM-before-step-INST-rc
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (INST-fetch-error-detected-p i))
                (not (b1p (flush-all? MA sigs)))
                (equal (INST-stg i) '(DQ 0))
                (not (b1p (dispatch-ready3? MA))))
           (equal (INST-tag (LRM-before (step-INST i MT MA sigs)
                                        (INST-rc i)
                                        (MT-step MT MA sigs)))
                  (dispatch-src3 MA)))
  :hints (("goal" :in-theory (enable dispatch-src3 DQ-out-tag3
                                     DISPATCH-READY3? lift-b-ops
                     MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                     INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                     INST-cntlv DQ-out-tag3 DQ-OUT-READY3?)
                :restrict
                ((MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                  ((i i)))
                 (INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                  ((i i)))))))

(defthm exist-LRM-before-p-if-BU-RS0
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(BU RS0))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (BU-RS-ready? (BU-RS0 (MA-BU MA)))))
                (MAETT-p MT) (MA-state-p MA))
           (exist-LRM-before-p i (INST-rc i) MT))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-BU-RS-P
```

```
                                    CONSISTENT-BU-RS0-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm execute-stg-p-LRM-before-if-BU-RS0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(BU RS0))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (BU-RS-ready? (BU-RS0 (MA-BU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (execute-stg-p (INST-stg (LRM-before i (INST-rc i) MT))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                    CONSISTENT-BU-RS-P
                                    CONSISTENT-BU-RS0-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm INST-tag-LRM-before-if-BU-RS0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(BU RS0))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (BU-RS-ready? (BU-RS0 (MA-BU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-tag (LRM-before i (INST-rc i) MT))
                    (BU-RS-src (BU-RS0 (MA-BU MA)))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                    CONSISTENT-BU-RS-P
                                    CONSISTENT-BU-RS0-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm exist-LRM-before-p-if-BU-RS1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(BU RS1))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (BU-RS-ready? (BU-RS1 (MA-BU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (exist-LRM-before-p i (INST-rc i) MT))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                    CONSISTENT-BU-RS-P
                                    CONSISTENT-BU-RS1-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm execute-stg-p-LRM-before-if-BU-RS1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(BU RS1))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (BU-RS-ready? (BU-RS1 (MA-BU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (execute-stg-p (INST-stg (LRM-before i (INST-rc i) MT))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                    CONSISTENT-BU-RS-P
                                    CONSISTENT-BU-RS1-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm INST-tag-LRM-before-if-BU-RS1
    (implies (and (inv MT MA)
```

```
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(BU RS1))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (BU-RS-ready? (BU-RS1 (MA-BU MA)))))
                (MAETT-p MT) (MA-state-p MA))
           (equal (INST-tag (LRM-before i (INST-rc i) MT))
                  (BU-RS-src (BU-RS1 (MA-BU MA)))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-BU-RS-P
                                     CONSISTENT-BU-RS1-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm exist-reg-ra-modifier-before-if-LSU-RS0
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(LSU RS0))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (LSU-RS-rdy1? (LSU-RS0 (MA-LSU MA)))))
                (MAETT-p MT) (MA-state-p MA))
           (exist-LRM-before-p i (INST-ra i) MT))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-LSU-RS-P
                                     CONSISTENT-LSU-RS0-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm execute-stg-p-last-reg-ra-modifier-before-if-LSU-RS0
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(LSU RS0))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (LSU-RS-rdy1? (LSU-RS0 (MA-LSU MA)))))
                (MAETT-p MT) (MA-state-p MA))
           (execute-stg-p (INST-stg (LRM-before i (INST-ra i) MT))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-LSU-RS-P
                                     CONSISTENT-LSU-RS0-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm INST-tag-last-reg-ra-modifier-before-if-LSU-RS0
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(LSU RS0))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (LSU-RS-rdy1? (LSU-RS0 (MA-LSU MA)))))
                (MAETT-p MT) (MA-state-p MA))
           (equal (INST-tag (LRM-before i (INST-ra i) MT))
                  (LSU-RS-src1 (LSU-RS0 (MA-LSU MA)))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-LSU-RS-P
                                     CONSISTENT-LSU-RS0-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm exist-reg-rb-modifier-before-if-LSU-RS0
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(LSU RS0))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
```

```
                (not (b1p (LSU-RS-rdy2? (LSU-RS0 (MA-LSU MA)))))
                (MAETT-p MT) (MA-state-p MA))
           (exist-LRM-before-p i (INST-rb i) MT))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-LSU-RS-P
                                     CONSISTENT-LSU-RS0-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm execute-stg-p-last-reg-rb-modifier-before-if-LSU-RS0
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(LSU RS0))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (LSU-RS-rdy2? (LSU-RS0 (MA-LSU MA)))))
                (MAETT-p MT) (MA-state-p MA))
           (execute-stg-p (INST-stg (LRM-before i (INST-rb i) MT))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-LSU-RS-P
                                     CONSISTENT-LSU-RS0-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm INST-tag-last-reg-rb-modifier-before-if-LSU-RS0
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(LSU RS0))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (LSU-RS-rdy2? (LSU-RS0 (MA-LSU MA)))))
                (MAETT-p MT) (MA-state-p MA))
           (equal (INST-tag (LRM-before i (INST-rb i) MT))
                  (LSU-RS-src2 (LSU-RS0 (MA-LSU MA)))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-LSU-RS-P
                                     CONSISTENT-LSU-RS0-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm exist-reg-rc-modifier-before-if-LSU-RS0
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(LSU RS0))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (LSU-RS-rdy3? (LSU-RS0 (MA-LSU MA)))))
                (MAETT-p MT) (MA-state-p MA))
           (exist-LRM-before-p i (INST-rc i) MT))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-LSU-RS-P
                                     CONSISTENT-LSU-RS0-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm execute-stg-p-last-reg-rc-modifier-before-if-LSU-RS0
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(LSU RS0))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (LSU-RS-rdy3? (LSU-RS0 (MA-LSU MA)))))
                (MAETT-p MT) (MA-state-p MA))
           (execute-stg-p (INST-stg (LRM-before i (INST-rc i) MT))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-LSU-RS-P
```

```
                                    CONSISTENT-LSU-RS0-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm INST-tag-last-reg-rc-modifier-before-if-LSU-RS0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU RS0))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (LSU-RS-rdy3? (LSU-RS0 (MA-LSU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-tag (LRM-before i (INST-rc i) MT))
                    (LSU-RS-src3 (LSU-RS0 (MA-LSU MA)))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                    CONSISTENT-LSU-RS-P
                                    CONSISTENT-LSU-RS0-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm exist-reg-ra-modifier-before-if-LSU-RS1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU RS1))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (LSU-RS-rdy1? (LSU-RS1 (MA-LSU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (exist-LRM-before-p i (INST-ra i) MT))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                    CONSISTENT-LSU-RS-P
                                    CONSISTENT-LSU-RS1-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm execute-stg-p-last-reg-ra-modifier-before-if-LSU-RS1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU RS1))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (LSU-RS-rdy1? (LSU-RS1 (MA-LSU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (execute-stg-p (INST-stg (LRM-before i (INST-ra i) MT))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                    CONSISTENT-LSU-RS-P
                                    CONSISTENT-LSU-RS1-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm INST-tag-last-reg-ra-modifier-before-if-LSU-RS1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU RS1))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (LSU-RS-rdy1? (LSU-RS1 (MA-LSU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-tag (LRM-before i (INST-ra i) MT))
                    (LSU-RS-src1 (LSU-RS1 (MA-LSU MA)))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                    CONSISTENT-LSU-RS-P
                                    CONSISTENT-LSU-RS1-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm exist-reg-rb-modifier-before-if-LSU-RS1
```

```
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU RS1))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (LSU-RS-rdy2? (LSU-RS1 (MA-LSU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (exist-LRM-before-p i (INST-rb i) MT))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-LSU-RS-P
                                     CONSISTENT-LSU-RS1-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm execute-stg-p-last-reg-rb-modifier-before-if-LSU-RS1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU RS1))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (LSU-RS-rdy2? (LSU-RS1 (MA-LSU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (execute-stg-p (INST-stg (LRM-before i (INST-rb i) MT))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-LSU-RS-P
                                     CONSISTENT-LSU-RS1-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm INST-tag-last-reg-rb-modifier-before-if-LSU-RS1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU RS1))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (LSU-RS-rdy2? (LSU-RS1 (MA-LSU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-tag (LRM-before i (INST-rb i) MT))
                    (LSU-RS-src2 (LSU-RS1 (MA-LSU MA)))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-LSU-RS-P
                                     CONSISTENT-LSU-RS1-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm exist-reg-rc-modifier-before-if-LSU-RS1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU RS1))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (LSU-RS-rdy3? (LSU-RS1 (MA-LSU MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (exist-LRM-before-p i (INST-rc i) MT))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-LSU-RS-P
                                     CONSISTENT-LSU-RS1-P)
                  :use (:instance consistent-RS-entry-p-inst-in))))

(defthm execute-stg-p-last-reg-rc-modifier-before-if-LSU-RS1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(LSU RS1))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
```

```
                (not (b1p (LSU-RS-rdy3? (LSU-RS1 (MA-LSU MA)))))
                (MAETT-p MT) (MA-state-p MA))
           (execute-stg-p (INST-stg (LRM-before i (INST-rc i) MT))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-LSU-RS-P
                                     CONSISTENT-LSU-RS1-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

(defthm INST-tag-last-reg-rc-modifier-before-if-LSU-RS1
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (equal (INST-stg i) '(LSU RS1))
                (not (b1p (INST-specultv? i)))
                (not (b1p (INST-modified? i)))
                (not (b1p (LSU-RS-rdy3? (LSU-RS1 (MA-LSU MA)))))
                (MAETT-p MT) (MA-state-p MA))
           (equal (INST-tag (LRM-before i (INST-rc i) MT))
                  (LSU-RS-src3 (LSU-RS1 (MA-LSU MA)))))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p
                                     CONSISTENT-LSU-RS-P
                                     CONSISTENT-LSU-RS1-P)
                :use (:instance consistent-RS-entry-p-inst-in))))

;;;; end of lemmas derived from consistent-RS-p

;;; Proof of consistent-RS-p for initial states
(defthm consistent-RS-p-init-MT
    (consistent-RS-p (init-MT MA) MA)
  :hints (("goal" :in-theory (enable init-MT consistent-RS-p))))

;;; invariant proof
; The proof approach is showing that each instruction satisfies
; consistent-RS-entry-p. Then prove consistent-RS-p by induction on
; instructions.

(defthm consistent-RS-entry-p-exintr-INST
    (consistent-RS-entry-p (exintr-INST MT ISA SMC) MT2 MA2)
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p))))

(defthm consistent-RS-entry-p-fetched-inst
    (consistent-RS-entry-p (fetched-inst MT ISA SPC SMC) MT2 MA2)
  :hints (("goal" :in-theory (enable consistent-RS-entry-p))))

; Minor case analysis follows.  Instead of explaining every case,
; I will explain only this case.
;
; Suppose dispatch-ready1? is 0 in the current cycle.  Instruction i
; is not speculatively executed, it is not in the self-modified
; instruction stream, or it has not caused any fetch error. (And
; flush-all? is 0, implying that no context switching takes place in
; this cycle.)  Since (logbit 0 (cntlv-operand (INST-cntlv i))) = 1,
; operands of instruction i are source register RA and RB.  If
; dispatch-ready1? is 0, the operand source register RA is not ready
; to supply the register value.  Thus, there should be a register
; modifier of register RA before instruction i, and this is true in
; the next state.
;
; The list of similar lemmas are:
;  exist-LRM-before-p-step-INST-operand0-rb
;  exist-LRM-before-p-step-INST-operand2
;  exist-LSRM-before-p-step-INST-operand3
;  exist-LRM-before-p-step-INST-rc
```

```
; exist-LRM-before-p-step-INST-operand2
(defthm exist-uncommitted-LRM-before-p-step-INST-operand0-ra
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (dispatch-ready1? MA)))
                  (b1p (logbit 0 (cntlv-operand (INST-cntlv i))))
                  (MAETT-p MT) (MA-state-p MA))
             (exist-uncommitted-LRM-before-p (step-INST i MT MA sigs)
                                             (INST-ra i)
                                             (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory
                  (enable dispatch-ready1? lift-b-ops
                          MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                          INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                          DQ-out-tag1 DQ-OUT-READY1?)
                  :restrict
                  ((MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                    ((i i)))
                   (INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                    ((i i)))))))

(defthm exist-uncommitted-LRM-before-p-step-INST-operand0-rb
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (dispatch-inst? MA))
                  (not (b1p (dispatch-ready2? MA)))
                  (MAETT-p MT) (MA-state-p MA))
             (exist-uncommitted-LRM-before-p (step-INST i MT MA sigs)
                                             (INST-rb i)
                                             (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory
                  (enable dispatch-ready2? lift-b-ops
                          MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                          INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                          DQ-out-tag2 DQ-OUT-READY2?)
                  :restrict
                  ((MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                    ((i i)))
                   (INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                    ((i i)))))))

(defthm exist-uncommitted-LRM-before-p-step-INST-operand2
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(DQ 0))
                  (b1p (dispatch-inst? MA))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (dispatch-ready1? MA)))
                  (b1p (logbit 2 (cntlv-operand (INST-cntlv i))))
```

```
                (MAETT-p MT) (MA-state-p MA))
           (exist-uncommitted-LRM-before-p (step-INST i MT MA sigs)
                                           (INST-rc i)
                                           (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory
                  (enable dispatch-ready1? lift-b-ops
                          MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                          INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                          DQ-out-tag1 DQ-OUT-READY1? INST-cntlv)
                  :restrict
                  ((MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                    ((i i)))
                   (INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                    ((i i)))))))

(defthm exist-uncommitted-LSRM-before-p-step-INST-operand3
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(DQ 0))
                  (b1p (dispatch-inst? MA))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (b1p (INST-decode-error? i)))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (dispatch-ready1? MA)))
                  (b1p (logbit 3 (cntlv-operand (INST-cntlv i))))
                  (MAETT-p MT) (MA-state-p MA))
             (exist-uncommitted-LSRM-before-p (step-INST i MT MA sigs)
                                              (INST-ra i)
                                              (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory
                  (enable dispatch-ready1? lift-b-ops
                          MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                          INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                          INST-cntlv
                          DQ-out-tag1 DQ-OUT-READY1?)
                  :restrict
                  ((MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                    ((i i)))
                   (INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                    ((i i)))))))

(defthm exist-uncommitted-LRM-before-p-step-INST-rc
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (dispatch-ready3? MA)))
                  (MAETT-p MT) (MA-state-p MA))
             (exist-uncommitted-LRM-before-p (step-INST i MT MA sigs)
                                             (INST-rc i)
                                             (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory
                  (enable dispatch-ready3? lift-b-ops
                          MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                          INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                          DQ-out-tag3 DQ-OUT-READY3?)
                  :restrict
```

```
                ((MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                  ((i i)))
                 (INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                  ((i i)))))))

; This lemmas and following similar lemmas are minor case analysis.  I
; will explain only this in detail Suppose instruction i is at the
; head of the dispatch queue.  And the control vector suggests that
; the operands for the instruction are RA and RB.  When instruction i
; is dispatched, dispatch-src1 provides Tomasulo's tag for the
; instruction that generates the source operand.  In fact, this
; indexes the ROB entry to which the last register modifier is
; assigned.  Since the last register modifier is the producer of the
; source operand, dispatch-src1 is correct.
;
; Similar lemmas are
;  INST-tag-LRM-before-step-INST-logbit0-rb
;  INST-tag-LRM-before-step-INST-logbit2
;  INST-tag-LSRM-before-step-INST-logbit3
(defthm INST-tag-LRM-before-step-INST-logbit0-ra
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (dispatch-ready1? MA)))
                  (b1p (logbit 0 (cntlv-operand (INST-cntlv i)))))
             (equal (INST-tag (LRM-before (step-INST i MT MA sigs)
                                          (INST-ra i)
                                          (MT-step MT MA sigs)))
                    (dispatch-src1 MA)))
  :hints (("goal" :in-theory (enable dispatch-src1 DQ-out-tag1
                                     DISPATCH-READY1? lift-b-ops
                     MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                     INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                     DQ-out-tag1 DQ-OUT-READY1?)
                  :restrict
                  ((MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                    ((i i)))
                   (INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                    ((i i)))))))

(defthm INST-tag-LRM-before-step-INST-logbit0-rb
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (dispatch-ready2? MA))))
             (equal (INST-tag (LRM-before (step-INST i MT MA sigs)
                                          (INST-rb i)
                                          (MT-step MT MA sigs)))
                    (dispatch-src2 MA)))
  :hints (("goal" :in-theory (enable dispatch-src2 DQ-out-tag2
                                     DISPATCH-READY2? lift-b-ops
                     MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
```

```
                   INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                   DQ-out-tag2 DQ-OUT-READY2?)
                  :restrict
                  ((MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                    ((i i)))
                   (INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                    ((i i)))))))

(defthm INST-tag-LRM-before-step-INST-logbit2
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (b1p (flush-all? MA sigs)))
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (dispatch-ready1? MA)))
                  (b1p (logbit 2 (cntlv-operand (INST-cntlv i)))))
             (equal (INST-tag (LRM-before (step-INST i MT MA sigs)
                                          (INST-rc i)
                                          (MT-step MT MA sigs)))
                    (dispatch-src1 MA)))
  :hints (("goal" :in-theory (enable dispatch-src1 DQ-out-tag1
                                     DISPATCH-READY1? lift-b-ops
                     MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                     INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                     INST-cntlv DQ-out-tag1 DQ-OUT-READY1?)
                  :restrict
                  ((MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                    ((i i)))
                   (INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                    ((i i)))))))

(defthm INST-tag-LSRM-before-step-INST-logbit3
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (b1p (INST-decode-error? i)))
                  (not (b1p (flush-all? MA sigs)))
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (dispatch-ready1? MA)))
                  (b1p (logbit 3 (cntlv-operand (INST-cntlv i)))))
             (equal (INST-tag (LSRM-before (step-INST i MT MA sigs)
                                           (INST-ra i)
                                           (MT-step MT MA sigs)))
                    (dispatch-src1 MA)))
  :hints (("goal" :in-theory (enable dispatch-src1 DQ-out-tag1
                                     DISPATCH-READY1? lift-b-ops
                     MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                     INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                     INST-cntlv DQ-out-tag1 DQ-OUT-READY1?)
                  :restrict
                  ((MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                    ((i i)))
                   (INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                    ((i i)))))))

; Following several lemmas are similar to each other.  I will only explain
```

```
; the first lemma.  If instruction i is at the head of dispatch queue, and
; its operand RA is not ready, then the last register modifier
; of register RA will still be in the execution stage in the next state.
;
; Note: If the register modifier completes its operation this cycle, its value
; should be on the CDB.  But dispatch-ready1? is 0 implies that it is not.
;
; Similar lemmas are proven in:
;  execute-stg-p-LRM-before-step-INST-if-logbit0-rb
;  execute-stg-p-LRM-before-step-INST-if-logbit2
;  execute-stg-p-LSRM-before-step-INST-if-logbit3
;  execute-stg-p-LRM-before-step-INST-rc
(defthm execute-stg-p-LRM-before-step-INST-if-logbit0-ra
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (b1p (flush-all? MA sigs)))
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (dispatch-ready1? MA)))
                  (b1p (logbit 0 (cntlv-operand (INST-cntlv i)))))
             (execute-stg-p (INST-stg (LRM-before (step-INST i MT MA sigs)
                                                  (INST-ra I)
                                                  (MT-step MT MA sigs))))))
  :hints (("Goal" :in-theory (enable dispatch-ready1? lift-b-ops
                                     DQ-out-tag1
                      MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                      INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                      DQ-out-ready1?))))

(defthm execute-stg-p-LRM-before-step-INST-if-logbit0-rb
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (b1p (flush-all? MA sigs)))
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (dispatch-ready2? MA))))
             (execute-stg-p
              (INST-stg (LRM-before (step-INST i MT MA sigs)
                                    (INST-rb I)
                                    (MT-step MT MA sigs)))))
  :hints (("Goal" :in-theory (enable dispatch-ready2? lift-b-ops
                                     DQ-out-tag2
                      MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                      INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                      DQ-out-ready2?))))

(defthm execute-stg-p-LRM-before-step-INST-if-logbit2
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (b1p (flush-all? MA sigs)))
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (dispatch-ready1? MA)))
```

```
                (b1p (logbit 2 (cntlv-operand (INST-cntlv i)))))
            (execute-stg-p (INST-stg (LRM-before (step-INST i MT MA sigs)
                                                 (INST-rc I)
                                                 (MT-step MT MA sigs)))))
  :hints (("Goal" :in-theory (enable dispatch-ready1? lift-b-ops
                                     DQ-out-tag1
                                     INST-cntlv
                     MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                     INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                     DQ-out-ready1?))))

(defthm execute-stg-p-LSRM-before-step-INST-if-logbit3
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (b1p (INST-decode-error? I)))
                  (not (b1p (flush-all? MA sigs)))
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (dispatch-ready1? MA)))
                  (b1p (logbit 3 (cntlv-operand (INST-cntlv i)))))
            (execute-stg-p (INST-stg (LSRM-before (step-INST i MT MA sigs)
                                                  (INST-ra I)
                                                  (MT-step MT MA sigs)))))
  :hints (("Goal" :in-theory (enable dispatch-ready1? lift-b-ops
                                     DQ-out-tag1
                                     INST-cntlv
                     MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                     INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                     DQ-out-ready1?))))

(defthm execute-stg-p-LRM-before-step-INST-rc
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (not (b1p (INST-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (not (INST-fetch-error-detected-p i))
                  (not (b1p (flush-all? MA sigs)))
                  (equal (INST-stg i) '(DQ 0))
                  (not (b1p (dispatch-ready3? MA))))
            (execute-stg-p (INST-stg (LRM-before (step-INST i MT MA sigs)
                                                 (INST-rc I)
                                                 (MT-step MT MA sigs)))))
  :hints (("Goal" :in-theory (enable dispatch-ready3? lift-b-ops
                                     DQ-out-tag3
                                     INST-cntlv
                     MT-SPECULTV-AT-DISPATCH-OFF-IF-NON-SPECULTV-INST-IN
                     INST-MODIFIED-AT-DISPATCH-OFF-IF-UNDISPATCHED-INST-IN
                     DQ-out-ready3?))))

; Proof of consistent-RS-entry-p-step-INST starts here.
; Following 28 lemmas are preparation for the final proof of
; consistent-RS-entry-p-step-INST, which is basically done by case
; analysis based on the current and next stage of i.
; For instance, consistent-IU-RS0-p-step-INST-DQ0 shows that
; consistent-IU-RS0-p is correct for i if its current stage is DQ0, and
; next stage is IU-RS0.
; Consistent-IU-RS0-p-step-INST proves consistent-IU-RS0-p for
; i whose next stage is IU-RS0, regardless of the current stage..
```

```
;
; Let me list all the 28 lemmas here.
;    consistent-IU-RS0-p-step-INST-DQ0
;    consistent-IU-RS0-p-step-INST-RS0
;   consistent-IU-RS0-p-step-INST
;    consistent-IU-RS1-p-step-INST-DQ0
;    consistent-IU-RS1-p-step-INST-RS1
;   consistent-IU-RS1-p-step-INST
; consistent-IU-RS-p-step-INST
;    consistent-MU-RS0-p-step-INST-DQ0
;    consistent-MU-RS0-p-step-INST-MU-RS0
;   consistent-MU-RS0-p-step-INST
;    consistent-MU-RS1-p-step-INST-DQ0
;    consistent-MU-RS1-p-step-INST-MU-RS1
;   consistent-MU-RS1-p-step-INST
; consistent-MU-RS-p-step-INST
;    consistent-BU-RS0-p-step-INST-DQ0
;    consistent-BU-RS0-p-step-INST-BU-RS0
;   consistent-BU-RS0-p-step-INST
;    consistent-BU-RS1-p-step-INST-DQ0
;    consistent-BU-RS1-p-step-INST-BU-RS1
;   consistent-BU-RS1-p-step-INST
; consistent-BU-RS-p-step-INST
;    consistent-LSU-RS0-p-step-INST-DQ0
;    consistent-LSU-RS0-p-step-INST-LSU-RS0
;   consistent-LSU-RS0-p-step-INST
;    consistent-LSU-RS1-p-step-INST-DQ0
;    consistent-LSU-RS1-p-step-INST-LSU-RS1
;   consistent-LSU-RS1-p-step-INST
; consistent-LSU-RS-p-step-INST

(defthm consistent-IU-RS0-p-step-INST-DQ0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(DQ 0))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(IU RS0))
                  (not (b1p (flush-all? MA sigs)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (consistent-IU-RS0-p (step-INST i MT MA sigs)
                                  (MT-step MT MA sigs)
                                  (MA-step MA sigs)))
  :hints (("Goal" :cases ((b1p (dispatch-to-IU? MA))))
          ("subgoal 2" :in-theory (enable step-inst-dq-inst
                                          step-inst-low-level-functions))
          ("subgoal 1" :in-theory (enable consistent-IU-RS0-p
                                          INST-STG-STEP-INST-IF-DISPATCH-IU
                                          dispatch-inst?
                                          step-IU step-IU-RS0
                                          lift-b-ops)
                       :cases ((b1p (INST-fetch-error? i))))
          ("subgoal 1.2" :cases ((b1p (INST-decode-error? i))))
          ("subgoal 1.2.1" :in-theory (enable dispatch-to-IU? lift-b-ops
                                              CONSISTENT-IU-RS0-P
                                              DQ-ready-to-IU?
                                              exception-relations
                                              INST-EXCPT-DETECTED-P))
          ("subgoal 1.1" :in-theory (enable dispatch-to-IU? lift-b-ops
                                            CONSISTENT-IU-RS0-P
                                            DQ-ready-to-IU?
                                            INST-EXCPT-DETECTED-P))))
```

```
(defthm consistent-IU-RS0-p-step-INST-RS0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (equal (INST-stg i) '(IU RS0))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(IU RS0))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (consistent-IU-RS0-p (step-INST i MT MA sigs)
                                  (MT-step MT MA sigs)
                                  (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable consistent-IU-RS0-p
                                     lift-b-ops
                                     SELECT-IU-RS0?
                                     DISPATCH-TO-IU?
                                     IU-READY?
                                     INST-STG-STEP-INST-IU-RS0
                                     step-IU step-IU-RS0))))

(defthm consistent-IU-RS0-p-step-INST
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(IU RS0))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (consistent-IU-RS0-p (step-INST i MT MA sigs)
                                  (MT-step MT MA sigs)
                                  (MA-step MA sigs)))
  :hints (("Goal" :use ((:instance stages-reachable-to-IU-RS0)))))

(defthm consistent-IU-RS1-p-step-INST-DQ0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(DQ 0))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(IU RS1))
                  (not (b1p (flush-all? MA sigs)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (consistent-IU-RS1-p (step-INST i MT MA sigs)
                                  (MT-step MT MA sigs)
                                  (MA-step MA sigs)))
  :hints (("Goal" :cases ((b1p (dispatch-to-IU? MA))))
          ("subgoal 2" :in-theory (enable step-inst-dq-inst
                                          step-inst-low-level-functions))
          ("subgoal 1" :in-theory (enable consistent-IU-RS1-p
                                          INST-STG-STEP-INST-IF-DISPATCH-IU
                                          dispatch-inst?
                                          SELECT-IU-RS0? SELECT-IU-RS1?
                                          DISPATCH-TO-IU? IU-READY?
                                          step-IU step-IU-RS1
                                          lift-b-ops)
                       :cases ((b1p (INST-fetch-error? i))))
          ("subgoal 1.2" :cases ((b1p (INST-decode-error? i))))
          ("subgoal 1.2.1" :in-theory (enable dispatch-to-IU? lift-b-ops
                                              CONSISTENT-IU-RS1-P
                                              DQ-ready-to-IU?
                                              exception-relations
                                              INST-EXCPT-DETECTED-P))
          ("subgoal 1.1" :in-theory (enable dispatch-to-IU? lift-b-ops
                                            CONSISTENT-IU-RS1-P
                                            DQ-ready-to-IU?
```

```
                                        INST-EXCPT-DETECTED-P))))

(defthm consistent-IU-RS1-p-step-INST-RS1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (equal (INST-stg i) '(IU RS1))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(IU RS1))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (consistent-IU-RS1-p (step-INST i MT MA sigs)
                                  (MT-step MT MA sigs)
                                  (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable consistent-IU-RS1-p
                                     lift-b-ops
                                     SELECT-IU-RS1?
                                     DISPATCH-TO-IU?
                                     IU-READY?
                                     INST-STG-STEP-INST-IU-RS1
                                     step-IU step-IU-RS1))))

(defthm consistent-IU-RS1-p-step-INST
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (equal (INST-stg (step-INST I MT MA sigs))
                         '(IU RS1))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (consistent-IU-RS1-p (step-INST i MT MA sigs)
                                  (MT-step MT MA sigs)
                                  (MA-step MA sigs)))
  :hints (("Goal" :use ((:instance stages-reachable-to-IU-RS1)))))

(defthm consistent-IU-RS-p-step-INST
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (IU-stg-p (INST-stg (step-INST i MT MA sigs)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (consistent-IU-RS-p (step-INST i MT MA sigs)
                                 (MT-step MT MA sigs)
                                 (MA-step MA sigs)))
  :hints (("Goal" :in-theory (enable consistent-IU-RS-p))))

(defthm consistent-MU-RS0-p-step-INST-DQ0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (equal (INST-stg i) '(DQ 0))
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(MU RS0))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (consistent-MU-RS0-p (step-INST i MT MA sigs)
                                  (MT-step MT MA sigs)
                                  (MA-step MA sigs)))
    :hints (("Goal" :cases ((b1p (dispatch-to-MU? MA))))
            ("subgoal 2" :in-theory (enable step-inst-dq-inst
                                            step-inst-low-level-functions))
            ("subgoal 1" :in-theory (enable consistent-MU-RS0-p
                                            INST-STG-STEP-INST-IF-DISPATCH-MU
                                            DQ-READY-TO-MU?
                                            dispatch-inst?
```

```
                                          DISPATCH-TO-MU? MU-READY?
                                          step-MU step-MU-RS0
                                          lift-b-ops)
                        :cases ((b1p (INST-fetch-error? i))))
            ("subgoal 1.2" :cases ((b1p (INST-decode-error? i))))
            ("subgoal 1.2.1" :in-theory (enable dispatch-to-MU? lift-b-ops
                                                CONSISTENT-MU-RS0-P
                                                DQ-READY-TO-MU?
                                                exception-relations
                                                INST-EXCPT-DETECTED-P))
            ("subgoal 1.1" :in-theory (enable dispatch-to-MU? lift-b-ops
                                              CONSISTENT-MU-RS0-P
                                              DQ-ready-to-MU?
                                              INST-EXCPT-DETECTED-P))))

(defthm consistent-MU-RS0-p-step-INST-MU-RS0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (equal (INST-stg i) '(MU RS0))
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(MU RS0))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (consistent-MU-RS0-p (step-INST i MT MA sigs)
                                  (MT-step MT MA sigs)
                                  (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable consistent-MU-RS0-p
                                     lift-b-ops
                                     SELECT-MU-RS0?
                                     DISPATCH-TO-MU?
                                     MU-READY?
                                     INST-STG-STEP-INST-MU-RS0
                                     step-MU step-MU-RS0))))

(defthm consistent-MU-RS0-p-step-INST
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(MU RS0))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (consistent-MU-RS0-p (step-INST i MT MA sigs)
                                  (MT-step MT MA sigs)
                                  (MA-step MA sigs)))
  :hints (("Goal" :use ((:instance stages-reachable-to-MU-RS0)))))

(defthm consistent-MU-RS1-p-step-INST-DQ0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (equal (INST-stg i) '(DQ 0))
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(MU RS1))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (consistent-MU-RS1-p (step-INST i MT MA sigs)
                                  (MT-step MT MA sigs)
                                  (MA-step MA sigs)))
    :hints (("Goal" :cases ((b1p (dispatch-to-MU? MA))))
            ("subgoal 2" :in-theory (enable step-inst-dq-inst
                                            step-inst-low-level-functions))
            ("subgoal 1" :in-theory (enable consistent-MU-RS1-p
                                            INST-STG-STEP-INST-IF-DISPATCH-MU
```

```
                                        DQ-READY-TO-MU?
                                        dispatch-inst?
                                        SELECT-MU-RS0? SELECT-MU-RS1?
                                        DISPATCH-TO-MU? MU-READY?
                                        step-MU step-MU-RS1
                                        lift-b-ops)
                        :cases ((b1p (INST-fetch-error? i))))
            ("subgoal 1.2" :cases ((b1p (INST-decode-error? i))))
            ("subgoal 1.2.1" :in-theory (enable dispatch-to-MU? lift-b-ops
                                                CONSISTENT-MU-RS1-P
                                                DQ-READY-TO-MU?
                                                exception-relations
                                                INST-EXCPT-DETECTED-P))
            ("subgoal 1.1" :in-theory (enable dispatch-to-MU? lift-b-ops
                                              CONSISTENT-MU-RS1-P
                                              DQ-ready-to-MU?
                                              INST-EXCPT-DETECTED-P))))

(defthm consistent-MU-RS1-p-step-INST-MU-RS1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (equal (INST-stg i) '(MU RS1))
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(MU RS1))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (consistent-MU-RS1-p (step-INST i MT MA sigs)
                                  (MT-step MT MA sigs)
                                  (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable consistent-MU-RS1-p
                                     lift-b-ops
                                     SELECT-MU-RS1?
                                     DISPATCH-TO-MU?
                                     MU-READY?
                                     INST-STG-STEP-INST-MU-RS1
                                     step-MU step-MU-RS1))))

(defthm consistent-MU-RS1-p-step-INST
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(MU RS1))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (consistent-MU-RS1-p (step-INST i MT MA sigs)
                                  (MT-step MT MA sigs)
                                  (MA-step MA sigs)))
  :hints (("Goal" :use ((:instance stages-reachable-to-MU-RS1)))))

(defthm consistent-MU-RS-p-step-INST
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (MU-stg-p (INST-stg (step-INST i MT MA sigs)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (consistent-MU-RS-p (step-INST i MT MA sigs)
                                 (MT-step MT MA sigs)
                                 (MA-step MA sigs)))
  :hints (("Goal" :in-theory (enable consistent-MU-RS-p))))

(defthm consistent-BU-RS0-p-step-INST-DQ0
    (implies (and (inv MT MA)
```

```
                (INST-in i MT) (INST-p i)
                (not (b1p (flush-all? MA sigs)))
                (equal (INST-stg i) '(DQ 0))
                (equal (INST-stg (step-INST i MT MA sigs)) '(BU RS0))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (consistent-BU-RS0-p (step-INST i MT MA sigs)
                                (MT-step MT MA sigs)
                                (MA-step MA sigs)))
  :hints (("Goal" :cases ((b1p (dispatch-to-BU? MA))))
          ("subgoal 2" :in-theory (enable step-inst-dq-inst
                                          step-inst-low-level-functions))
          ("subgoal 1" :in-theory (enable consistent-BU-RS0-p
                                          INST-STG-STEP-INST-IF-DISPATCH-BU
                                          DQ-READY-TO-BU?
                                          dispatch-inst?
                                          SELECT-BU-RS0? SELECT-BU-RS1?
                                          DISPATCH-TO-BU? BU-READY?
                                          step-BU step-BU-RS0
                                          lift-b-ops)
                       :cases ((b1p (INST-fetch-error? i))))
          ("subgoal 1.2" :cases ((b1p (INST-decode-error? i))))
          ("subgoal 1.2.1" :in-theory (enable dispatch-to-BU? lift-b-ops
                                              CONSISTENT-BU-RS0-P
                                              DQ-READY-TO-BU?
                                              exception-relations
                                              INST-EXCPT-DETECTED-P))
          ("subgoal 1.1" :in-theory (enable dispatch-to-BU? lift-b-ops
                                            CONSISTENT-BU-RS0-P
                                            DQ-ready-to-BU?
                                            INST-EXCPT-DETECTED-P))))

(defthm consistent-BU-RS0-p-step-INST-BU-RS0
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (not (b1p (flush-all? MA sigs)))
                (equal (INST-stg i) '(BU RS0))
                (equal (INST-stg (step-INST i MT MA sigs)) '(BU RS0))
                (MAETT-p MT) (MA-state-p MA)  (MA-input-p sigs))
           (consistent-BU-RS0-p (step-INST i MT MA sigs)
                                (MT-step MT MA sigs)
                                (MA-step MA sigs)))
    :hints (("goal" :in-theory (enable consistent-BU-RS0-p
                                       lift-b-ops
                                       SELECT-BU-RS0?
                                       DISPATCH-TO-BU?
                                       BU-READY?
                                       INST-STG-STEP-INST-BU-RS0
                                       step-BU step-BU-RS0))))

(defthm consistent-BU-RS0-p-step-INST
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (not (b1p (flush-all? MA sigs)))
                (equal (INST-stg (step-INST i MT MA sigs)) '(BU RS0))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (consistent-BU-RS0-p (step-INST i MT MA sigs)
                                (MT-step MT MA sigs)
                                (MA-step MA sigs)))
    :hints (("Goal" :use ((:instance stages-reachable-to-BU-RS0)))))

(defthm consistent-BU-RS1-p-step-INST-DQ0
    (implies (and (inv MT MA)
```

```
                (INST-in i MT) (INST-p i)
                (not (b1p (flush-all? MA sigs)))
                (equal (INST-stg i) '(DQ 0))
                (equal (INST-stg (step-INST i MT MA sigs)) '(BU RS1))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (consistent-BU-RS1-p (step-INST i MT MA sigs)
                                (MT-step MT MA sigs)
                                (MA-step MA sigs)))
    :hints (("Goal" :cases ((b1p (dispatch-to-BU? MA))))
            ("subgoal 2" :in-theory (enable step-inst-dq-inst
                                            step-inst-low-level-functions))
            ("subgoal 1" :in-theory (enable consistent-BU-RS1-p
                                            INST-STG-STEP-INST-IF-DISPATCH-BU
                                            DQ-READY-TO-BU?
                                            dispatch-inst?
                                            SELECT-BU-RS0? SELECT-BU-RS1?
                                            DISPATCH-TO-BU? BU-READY?
                                            step-BU step-BU-RS1
                                            lift-b-ops)
                          :cases ((b1p (INST-fetch-error? i))))
            ("subgoal 1.2" :cases ((b1p (INST-decode-error? i))))
            ("subgoal 1.2.1" :in-theory (enable dispatch-to-BU? lift-b-ops
                                                CONSISTENT-BU-RS1-P
                                                DQ-READY-TO-BU?
                                                exception-relations
                                                INST-EXCPT-DETECTED-P))
            ("subgoal 1.1" :in-theory (enable dispatch-to-BU? lift-b-ops
                                              CONSISTENT-BU-RS1-P
                                              DQ-ready-to-BU?
                                              INST-EXCPT-DETECTED-P))))

(defthm consistent-BU-RS1-p-step-INST-BU-RS1
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (not (b1p (flush-all? MA sigs)))
                (equal (INST-stg i) '(BU RS1))
                (equal (INST-stg (step-INST i MT MA sigs)) '(BU RS1))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (consistent-BU-RS1-p (step-INST i MT MA sigs)
                                (MT-step MT MA sigs)
                                (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable consistent-BU-RS1-p
                                     lift-b-ops
                                     SELECT-BU-RS1?
                                     DISPATCH-TO-BU?
                                     BU-READY?
                                     INST-STG-STEP-INST-BU-RS1
                                     step-BU step-BU-RS1))))

(defthm consistent-BU-RS1-p-step-INST
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (not (b1p (flush-all? MA sigs)))
                (equal (INST-stg (step-INST i MT MA sigs)) '(BU RS1))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (consistent-BU-RS1-p (step-INST i MT MA sigs)
                                (MT-step MT MA sigs)
                                (MA-step MA sigs)))
     :hints (("Goal" :use ((:instance stages-reachable-to-BU-RS1)))))

(defthm consistent-BU-RS-p-step-INST
    (implies (and (inv MT MA)
```

```
                (INST-in i MT) (INST-p i)
                (not (b1p (flush-all? MA sigs)))
                (BU-stg-p (INST-stg (step-INST i MT MA sigs)))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (consistent-BU-RS-p (step-INST i MT MA sigs)
                               (MT-step MT MA sigs)
                               (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable consistent-BU-RS-p))))

(defthm consistent-LSU-RS0-p-step-INST-DQ0
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (not (b1p (flush-all? MA sigs)))
                (equal (INST-stg i) '(DQ 0))
                (equal (INST-stg (step-INST i MT MA sigs))
                       '(LSU RS0))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (consistent-LSU-RS0-p (step-INST i MT MA sigs)
                                 (MT-step MT MA sigs)
                                 (MA-step MA sigs)))
  :hints (("Goal" :cases ((b1p (dispatch-to-LSU? MA))))
          ("subgoal 2" :in-theory (enable step-inst-dq-inst
                                          DISPATCH-INST? lift-b-ops
                                          step-inst-low-level-functions))
          ("subgoal 1" :in-theory
                       (e/d  (consistent-LSU-RS0-p
                              INST-STG-STEP-INST-IF-DISPATCH-LSU
                              DQ-READY-TO-LSU?
                              dispatch-inst?
                              SELECT-LSU-RS0? SELECT-LSU-RS1?
                              DISPATCH-TO-LSU? LSU-READY?
                              step-LSU step-LSU-RS0
                              lift-b-ops)
                             (DQ-DE0-CNTLV-=-INST-CNTLV-2
                              DQ-DE0-EXCPT-=-INST-EXCPT-FLAGS-2
                              INST-SPECULTV-INST-AT-DQ0-IF-DISPATCH-INST
                              UNIQ-INST-AT-DQ0-IF-DISPATCH-INST))
                       :cases ((b1p (INST-fetch-error? i))))
          ("subgoal 1.2" :cases ((b1p (INST-decode-error? i))))
          ("subgoal 1.2.1" :in-theory (enable dispatch-to-LSU? lift-b-ops
                                              CONSISTENT-LSU-RS0-P
                                              DQ-READY-TO-LSU?
                                              exception-relations
                                              INST-EXCPT-DETECTED-P))
          ("subgoal 1.1" :in-theory (enable dispatch-to-LSU? lift-b-ops
                                            CONSISTENT-LSU-RS0-P
                                            DQ-ready-to-LSU?
                                            INST-EXCPT-DETECTED-P))))

(defthm consistent-LSU-RS0-p-step-INST-LSU-RS0
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (not (b1p (flush-all? MA sigs)))
                (equal (INST-stg i) '(LSU RS0))
                (equal (INST-stg (step-INST i MT MA sigs))
                       '(LSU RS0))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (consistent-LSU-RS0-p (step-INST i MT MA sigs)
                                 (MT-step MT MA sigs)
                                 (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable consistent-LSU-RS0-p
                                     lift-b-ops
```

```
                                    SELECT-LSU-RS0?
                                    DISPATCH-TO-LSU?
                                    LSU-READY?
                                    INST-STG-STEP-INST-LSU-RS0
                                    step-LSU step-LSU-RS0))))

(defthm consistent-LSU-RS0-p-step-INST
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU RS0))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (consistent-LSU-RS0-p (step-INST i MT MA sigs)
                                   (MT-step MT MA sigs)
                                   (MA-step MA sigs)))
  :Hints (("goal" :use (:instance stages-reachable-to-LSU-RS0))))

(defthm consistent-LSU-RS1-p-step-INST-DQ0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
                  (equal (INST-stg i) '(DQ 0))
                  (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU RS1))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (consistent-LSU-RS1-p (step-INST i MT MA sigs)
                                   (MT-step MT MA sigs)
                                   (MA-step MA sigs)))
  :hints (("Goal" :cases ((b1p (dispatch-to-LSU? MA))))
          ("subgoal 2" :in-theory (enable step-inst-dq-inst
                                          DISPATCH-INST? lift-b-ops
                                          step-inst-low-level-functions))
          ("subgoal 1" :in-theory
                       (e/d  (consistent-LSU-RS1-p
                              INST-STG-STEP-INST-IF-DISPATCH-LSU
                              DQ-READY-TO-LSU?
                              dispatch-inst?
                              SELECT-LSU-RS0? SELECT-LSU-RS1?
                              DISPATCH-TO-LSU? LSU-READY?
                              step-LSU step-LSU-RS1
                              lift-b-ops)
                             (DQ-DE0-CNTLV-=-INST-CNTLV-2
                              DQ-DE0-EXCPT-=-INST-EXCPT-FLAGS-2
                              INST-SPECULTV-INST-AT-DQ0-IF-DISPATCH-INST
                              UNIQ-INST-AT-DQ0-IF-DISPATCH-INST))
                       :cases ((b1p (INST-fetch-error? i))))
          ("subgoal 1.2" :cases ((b1p (INST-decode-error? i))))
          ("subgoal 1.2.1" :in-theory (enable dispatch-to-LSU? lift-b-ops
                                              CONSISTENT-LSU-RS1-P
                                              DQ-READY-TO-LSU?
                                              exception-relations
                                              INST-EXCPT-DETECTED-P))
          ("subgoal 1.1" :in-theory (enable dispatch-to-LSU? lift-b-ops
                                            CONSISTENT-LSU-RS1-P
                                            DQ-ready-to-LSU?
                                            INST-EXCPT-DETECTED-P))))

(defthm consistent-LSU-RS1-p-step-INST-LSU-RS1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (flush-all? MA sigs)))
```

```
                (equal (INST-stg i) '(LSU RS1))
                (equal (INST-stg (step-INST i MT MA sigs))
                       '(LSU RS1))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (consistent-LSU-RS1-p (step-INST i MT MA sigs)
                                 (MT-step MT MA sigs)
                                 (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable consistent-LSU-RS1-p
                                     lift-b-ops
                                     SELECT-LSU-RS1?
                                     DISPATCH-TO-LSU?
                                     LSU-READY?
                                     INST-STG-STEP-INST-LSU-RS1
                                     step-LSU step-LSU-RS1))))

(defthm consistent-LSU-RS1-p-step-INST
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (not (b1p (flush-all? MA sigs)))
                (equal (INST-stg (step-INST i MT MA sigs))
                       '(LSU RS1))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (consistent-LSU-RS1-p (step-INST i MT MA sigs)
                                 (MT-step MT MA sigs)
                                 (MA-step MA sigs)))
  :Hints (("goal" :use (:instance stages-reachable-to-LSU-RS1))))

(defthm consistent-LSU-RS-p-step-INST
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (not (b1p (flush-all? MA sigs)))
                (LSU-stg-p (INST-stg (step-INST i MT MA sigs)))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (consistent-LSU-RS-p (step-INST i MT MA sigs)
                                (MT-step MT MA sigs)
                                (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable consistent-LSU-RS-p))))

; A landmark lemma.
; A consistent-RS-entry-p is true for the instruction i in the next
; state.
(defthm consistent-RS-entry-p-step-INST
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (not (b1p (flush-all? MA sigs)))
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (consistent-RS-entry-p (step-INST i MT MA sigs)
                                  (MT-step MT MA sigs)
                                  (MA-step MA sigs)))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p))))

(defthm trace-consistent-RS-p-step-trace
    (implies (and (inv MT MA)
                (not (b1p (flush-all? MA sigs)))
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
           (trace-consistent-RS-p (step-trace trace MT MA sigs ISA spc smc)
                                  (MT-step MT MA sigs)
                                  (MA-step MA sigs))))

; a help lemma.
(defthm consistent-RS-entry-p-if-committed-p
```

```
    (implies (and (INST-p i) (committed-p i))
             (consistent-RS-entry-p i MT MA))
  :hints (("Goal" :in-theory (enable consistent-RS-entry-p committed-p
                                     EXECUTE-STG-P))))

(defthm trace-consistent-RS-p-step-trace-if-flush-all
    (implies (and (inv MT MA)
                  (b1p (flush-all? MA sigs))
                  (MT-all-commit-before-trace trace MT)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (trace-consistent-RS-p (step-trace trace MT MA sigs ISA spc smc)
                                    (MT-step MT MA sigs)
                                    (MA-step MA sigs)))
  :hints ((when-found (MT-ALL-COMMIT-BEFORE-TRACE (CDR TRACE) MT)
                      (:cases ((committed-p (car trace)))))))

;; consistent-RS-p is preserved during the MA transition.
(defthm consistent-RS-p-preserved
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (consistent-RS-p (MT-step MT MA sigs) (MA-step MA sigs)))
  :hints (("Goal" :in-theory (enable consistent-RS-p )
                  :cases ((b1p (flush-all? MA sigs))))))
```

D.6.7 ISA-comp.lisp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; File: ISA-comp.lisp
; Author  Jun Sawada, University of Texas at Austin
;
; This file includes the proof of the invariant properties pc-match-p,
; RF-match-p, SRF-match-p, and mem-match-p.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(in-package "ACL2")

(include-book "MA2-lemmas")
(include-book "MAETT-lemmas")
(include-book "memory-inv")

(deflabel begin-ISA-comp)
; This file proves the invariant conditions specifying the correct
; state of programmer visible states.
;  Index
;   Misc Lemmas
;   Proof for RF-match-p
;      Base case
;       RF-match-p-init-MA
;      Induction case
;        Lemmas for no state changes
;        MT-RF-ISA-RF-inst-of-tag
;           ISA-RF-ISA-step-if-INST-wb
;           ROB-write-val-INST-dest-val
;           ROB-write-rid-INST-dest-reg
;           step-RF-INST-post-ISA-of-INST-at-MT-ROB-head-normal-case
;        step-RF-INST-post-ISA-of-INST-at-MT-ROB-head
;       RF-match-p-preserved
;
;   Proof of SRF-Match
```

```
;      Base case
;        SRF-match-p-init-MA
;      Induction case
;        Case 1
;         step-SRF-if-ex-intr?
;         MT-SRF-MT-step-if-ex-intr
;         ISA-SRF-ISA-step-exintr
;        Case 2
;         step-SRF-if-not-commit-inst?
;         MT-SRF-MT-step-if-not-commit-inst
;        Case 3
;         MT-SRF-ISA-RF-inst-of-tag
;         step-SRF-INST-post-ISA-of-INST-at-MT-ROB-head
;       SRF-match-p-preserved
;
;   Proof of pc-match-p preserved
;     Base case
;       pc-match-p-init-MT
;     Induction step
;         Case when ex-intr? is 1.
;           MT-pc-rob-jmp-addr-if-ex-intr
;         Case when enter-excpt? is 1
;           MT-pc-rob-jmp-addr-if-enter-excpt
;         Case when commit-jmp? is on
;           MT-pc-rob-jmp-addr-if-commit-jmp
;         Case when leave-excpt? is on
;           MT-pc-if-leave-excpt
;         Case when fetch-inst? is on
;           MT-pc-if-fetch-inst
;         Otherwise
;       pc-match-p-preserved
;
;   Proof of mem-match-p
;     Base Case
;       mem-match-p-init-MT
;     Induction Case
;         MT-mem-if-step-MT-if-release-wbuf0
;         MT-mem-if-step-MT-if-not-release-wbuf0
;       mem-match-p-preserved
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Misc Lemmas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(local
(defthm trace-SRF*
    (equal (trace-SRF INST-list SRF)
           (if (endp INST-list)
               SRF
               (if (not (committed-p (car INST-list)))
                   SRF
                   (trace-SRF (cdr INST-list)
                              (ISA-SRF (INST-post-ISA (car INST-list)))))))
  :rule-classes :definition))

(local (in-theory (disable trace-SRF* )))

;; Several lemmas given below are for the proof of lemmas in this book.
;; They are not intended to be a universal lemma, because the left-hand
;; side terms are not definitely more complex than right-hand side.
;; However, these rewriting rules are useful in the proof of lemmas in
;; this book.
(local
```

```
(defthm MT-mem-=-MA-mem
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA))
             (equal (MT-mem MT) (MA-mem MA)))
  :hints (("goal" :in-theory (enable inv mem-match-p)))))

(local
(defthm MT-RF-=-MA-RF
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA))
             (equal (MT-RF MT) (MA-RF MA)))
  :hints (("goal" :in-theory (enable inv RF-match-p)))))

(local
(defthm MT-SRF-=-MA-SRF
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA))
             (equal (MT-SRF MT) (MA-SRF MA)))
  :hints (("goal" :in-theory (enable inv SRF-match-p)))))

(local
(defthm MT-pc-=-MA-pc
    (implies (and (inv MT MA)
                  (not (b1p (MT-specultv? MT)))
                  (not (b1p (MT-self-modify? MT)))
                  (MAETT-p MT)
                  (MA-state-p MA))
             (equal (MT-pc MT) (MA-pc MA)))
  :hints (("goal" :in-theory (enable inv
                                     MT-specultv-p MT-self-modify-p
                                     pc-match-p lift-b-ops)))))

(encapsulate nil
(local
(defthm ISA-pc-MT-final-ISA-=-MT-pc-help
    (equal (ISA-pc (trace-final-ISA trace ISA))
           (trace-pc trace (ISA-pc ISA)))))

;  The program counter contains the value of the PC in the final ISA state.
(defthm ISA-pc-MT-final-ISA-=-MT-pc
    (equal (ISA-pc (MT-final-ISA MT)) (MT-pc MT))
  :hints (("goal" :in-theory (enable MT-final-ISA MT-pc))))
)

(local
 (defthm ISA-step-INST-pre-ISA
     (implies (and (not (b1p (ISA-input-exint intr)))
                   (not (b1p (INST-fetch-error? i))))
              (equal (ISA-step (INST-pre-ISA i) intr)
                     (LET ((S (INST-pre-ISA i))
                           (INST (READ-MEM (ISA-PC (INST-pre-ISA i))
                                           (ISA-MEM (INST-pre-ISA i)))))
                       (LET ((OP (OPCODE INST))
                             (RC (RC INST))
                             (RA (RA INST))
                             (RB (RB INST))
                             (IM (IM INST)))
                         (COND ((EQUAL OP 0) (ISA-ADD RC RA RB S))
                               ((EQUAL OP 1) (ISA-MUL RC RA RB S))
```

```
                                ((EQUAL OP 2) (ISA-BR RC IM S))
                                ((EQUAL OP 3) (ISA-LD RC RA RB S))
                                ((EQUAL OP 6) (ISA-LDI RC IM S))
                                ((EQUAL OP 4) (ISA-ST RC RA RB S))
                                ((EQUAL OP 7) (ISA-STI RC IM S))
                                ((EQUAL OP 5) (ISA-SYNC S))
                                ((EQUAL OP 8) (ISA-RFEH S))
                                ((EQUAL OP 9) (ISA-MFSR RC RA S))
                                ((EQUAL OP 10) (ISA-MTSR RC RA S))
                                (T (ISA-ILLEGAL-INST s)))))))
   :Hints (("goal" :in-theory (enable ISA-step INST-fetch-error?
                                      lift-b-ops)))))
(local (in-theory (disable ISA-step-INST-pre-ISA)))

(encapsulate nil
(local
(defthm MT-CMI-p-MT-step-if-INST-at-ROB-head-modified-induct
    (implies (and (b1p (INST-modified? i))
                  (INST-p i) (member-equal i trace)
                  (trace-all-commit-before i trace)
                  (dispatched-p i)
                  (committed-p (step-INST i MT MA sigs))
                  (INST-listp trace))
             (trace-CMI-p (step-trace trace MT MA sigs
                                      ISA spc smc)))))

(local
(defthm MT-CMI-p-MT-step-if-INST-at-ROB-head-modified-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (INST-modified? i))
                  (INST-p i) (INST-in i MT)
                  (MT-all-commit-before i MT)
                  (dispatched-p i)
                  (committed-p (step-INST i MT MA sigs)))
             (MT-CMI-p (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable MT-CMI-p INST-in
                                     MT-step MT-all-commit-before)))))

; If instruction at the head of the ROB is modified and commit-inst? is on,
; commitment of a self-modified instruction happens.
(defthm MT-CMI-p-MT-step-if-INST-at-ROB-head-modified
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (commit-inst? MA))
                  (b1p (INST-modified? (inst-of-tag (MT-ROB-head MT) MT))))
             (MT-CMI-p (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (e/d (commit-inst? lift-b-ops)
                                  (incompatible-with-excpt-in-MAETT-lemmas)))))
)

(encapsulate nil
(local
(defthm MT-self-modified-p-MT-step-if-INST-at-ROB-head-modified-induct
    (implies (and (b1p (INST-modified? i))
                  (INST-p i) (member-equal i trace)
                  (trace-all-commit-before i trace)
                  (dispatched-p i)
                  (committed-p (step-INST i MT MA sigs))
                  (INST-listp trace))
             (equal (trace-self-modify? (step-trace trace MT MA sigs
                                                    ISA spc smc))
```

```
                  1))
  :hints (("goal" :in-theory (enable lift-b-ops)))))

(local
(defthm MT-CMI-p-MT-step-if-INST-at-ROB-head-modified-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (INST-modified? i))
                  (INST-p i) (INST-in i MT)
                  (MT-all-commit-before i MT)
                  (dispatched-p i)
                  (committed-p (step-INST i MT MA sigs)))
             (equal (MT-self-modify? (MT-step MT MA sigs)) 1))
  :hints (("goal" :in-theory (enable MT-self-modify? INST-in
                                     MT-step MT-all-commit-before)))))

; If instruction at the head of the ROB is modified and commit-inst? is on,
; a self-modified instruction is committed.
(defthm MT-self-modify-MT-step-if-INST-at-ROB-head-modified
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (commit-inst? MA))
                  (b1p (INST-modified? (inst-of-tag (MT-ROB-head MT) MT))))
             (equal (MT-self-modify? (MT-step MT MA sigs)) 1))
  :hints (("goal" :in-theory (e/d (commit-inst? lift-b-ops)
                                  (incompatible-with-excpt-in-MAETT-lemmas)))))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Proof of invariant RF-match-p
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;; Proof of match-RF-p for the initial case
(defthm RF-match-p-init-MA
    (implies (MA-state-p MA)
             (RF-match-p (init-MT MA) MA))
  :Hints (("goal" :in-theory (enable RF-match-p init-MT
                                     MT-RF))))

;;;;  Induction case.
; First we prove basic lemmas to characterize the case where no
; changes occur to the register files.  Then we prove that the state
; changes on the register file in the "actual" machine state, and the
; "ideal" machine state calculated from the intermediate abstraction.

;;;;;  Lemmas to characterize when the register file does not change.

; If no instruction commits in this cycle, the register file is not updated.
(defthm commit-inst-if-rob-write-reg
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (commit-inst? MA))))
             (equal (ROB-write-reg? (MA-rob MA))  0))
  :hints (("goal" :in-theory (enable commit-inst? ROB-write-reg?
                                     lift-b-ops equal-b1p-converter))))

; If no instruction commits, the register file is not modified.
(defthm step-RF-if-not-commit-inst?
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (commit-inst? MA))))
             (equal (step-RF MA) (MA-RF MA)))
```

```
  :hints (("goal" :in-theory (enable step-RF))))

; If external interrupt to the ISA does not modify the general register file.
(defthm ISA-RF-ISA-step-with-exintr
    (implies (b1p (ISA-input-exint intr))
             (equal (ISA-RF (ISA-step ISA intr))
                    (ISA-RF ISA)))
  :hints (("goal" :in-theory (enable ISA-def))))

(encapsulate nil
(local
(defthm MT-RF-MT-step-if-not-commit-inst-help
    (implies (not (b1p (commit-inst? MA)))
             (equal (trace-RF (step-trace trace MT MA sigs
                                          ISA spc smc)
                              (ISA-RF ISA))
                    (trace-RF trace (ISA-RF ISA))))))

; If no instruction commits this cycle, MT-RF remains identical.
(defthm MT-RF-MT-step-if-not-commit-inst
    (implies (not (b1p (commit-inst? MA)))
             (equal (MT-RF (MT-step MT MA sigs))
                    (MT-RF MT)))
  :hints (("goal" :in-theory (enable MT-RF MT-step))))
)


(encapsulate nil
(local
(defthm committed-p-step-inst-if-commit-inst?
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (complete-stg-p
                   (INST-stg (inst-of-tag (MT-rob-head MT) MT)))
                  (b1p (commit-inst? MA)))
             (committed-p (step-INST (inst-of-tag (MT-rob-head MT) MT)
                                     MT MA sigs)))
  :hints (("goal" :in-theory (enable step-inst-opener
                                     INST-commit?
                                     lift-b-ops
                                     bv-eqv-iff-equal
                                     step-INST-complete)))))

; The instruction at the head of the ROB commits if commit-inst? is on.
(defthm  committed-p-step-inst-INST-at-MT-rob-head
    (implies (and (inv MT MA)
                  (MA-state-p MA) (MAETT-p MT)
                  (b1p (commit-inst? MA)))
             (committed-p (step-INST  (inst-of-tag (MT-ROB-head MT) MT)
                                      MT MA sigs)))
  :hints (("goal" :use (:instance INST-is-at-one-of-the-stages
                                  (i (inst-of-tag (MT-ROB-head MT) MT)))
                  :in-theory (disable INST-is-at-one-of-the-stages))
          ("subgoal 2" :by nil))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
    (implies
     (and (inv MT MA)
          (MA-state-p MA) (MAETT-p MT) (b1p (commit-inst? MA))
          (not (commit-stg-p (INST-stg
                              (step-INST (inst-of-tag (MT-ROB-head MT) MT)
```

```
                                        MT MA sigs)))))
     (retire-stg-p (INST-stg
                    (step-INST (inst-of-tag (MT-ROB-head MT) MT) MT MA sigs))))
    :hints (("goal" :in-theory
                    (e/d (committed-p)
                         (NOT-COMMITTED-P-IF-NOT-COMMIT-RETIRE)))))))
)

;;;;;;;  Lemmas to compare the state change on register file
;;;;;;;  in the actual implementation and the ideal state.
;
; The proof of RF-match-p-preserved consists of two major
; lemmas: step-RF-INST-post-ISA-of-INST-at-MT-ROB-head and
; MT-RF-ISA-RF-inst-of-tag.
;
; step-RF-INST-post-ISA-of-INST-at-MT-ROB-head tells
; that the register file in the next "actual machine state" is the same as
; in the post-ISA of the instruction at the head of the ROB.
;
; MT-RF-ISA-RF-inst-of-tag
;  calculates what MT-RF calculates as the "ideal register file"
;
;;;;;;;
(local
(defthm not-retire-stg-p-step-inst-if-earlier-inst-commit
    (implies (and (inv MT MA)
                  (b1p (INST-commit? j MA))
                  (INST-in-order-p j i MT)
                  (INST-in i MT) (INST-p i)
                  (INST-in j MT) (INST-p j)
                  (MAETT-p MT) (MA-state-p MA))
             (not (committed-p (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (e/d (committed-p)
                                  (complete-stg-p-if-INST-commit
                                   INST-is-at-one-of-the-stages))
                  :use ((:instance complete-stg-p-if-INST-commit (i j))
                        (:instance INST-is-at-one-of-the-stages))))))

; local lemmas.
; committed-p-step-inst-car says:
;   If j commits, (car trace) has already committed.
; not-INST-cause-jmp-car-when-inst-commit states:
;   If j commits, (car trace) does not commit in this cycle.
;
(local
(defthm committed-p-step-inst-car
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (member-equal j (cdr trace))
                  (b1p (INST-commit? j MA))
                  (INST-listp trace) (INST-in j MT) (INST-p j)
                  (MAETT-p MT) (MA-state-p MA))
             (committed-p (step-INST (car trace) MT MA sigs)))
  :hints (("goal" :use (:instance committed-p-if-INST-commit-following
                                  (i (car trace)))
                  :in-theory (enable committed-p)))))

(local
(defthm not-INST-cause-jmp-car-when-inst-commit
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (member-equal j (cdr trace))
```

```
                (b1p (INST-commit? j MA))
                (INST-listp trace)
                (INST-in j MT) (INST-p j)
                (MAETT-p MT) (MA-state-p MA))
           (equal (INST-cause-jmp? (car trace) MT MA sigs) 0))
  :hints (("goal" :in-theory (e/d (committed-p INST-commit? lift-b-ops
                                               equal-b1p-converter
                                               INST-cause-jmp?)
                                  (complete-stg-p-if-INST-commit
                                   INST-is-at-one-of-the-stages))
                :use ((:instance complete-stg-p-if-INST-commit (i j))
                      (:instance INST-is-at-one-of-the-stages))))))

(encapsulate nil
(local
(defthm MT-RF-MT-step-if-INST-commit-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT)
                (INST-listp trace)
                (member-equal i trace)
                (b1p (INST-commit? i MA))
                (INST-p i) (MAETT-p MT) (MA-state-p MA))
           (equal (trace-RF (step-trace trace MT MA sigs
                                        ISA spc smc)
                            (ISA-RF (INST-pre-ISA (car trace))))
                  (ISA-RF (INST-post-ISA i))))
  :rule-classes nil
  :hints (("goal" :in-theory (e/d (trace-RF*
                                   INST-exintr-now-INST-commit-exclusive)
                                  (trace-RF))
                :induct t)
          (when-found (INST-pre-ISA (car (cdr trace)))
                      (:cases ((consp (cdr trace)))))
          (when-found (consp (cdr trace))
                      (:expand
                       (STEP-TRACE (CDR TRACE)
                           MT MA SIGS
                           (ISA-STEP (INST-PRE-ISA (CAR TRACE))
                                     (ISA-input (INST-EXINTR? (CAR TRACE))))
                           (B-IOR (INST-SPECULTV? (CAR TRACE))
                                  (INST-START-SPECULTV? (CAR TRACE)))
                           (INST-MODIFIED? (CAR TRACE)))))))))

(local
(defthm MT-RF-MT-step-if-INST-commit?
    (implies (and (inv MT MA)
                (b1p (INST-commit? i MA))
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA))
           (equal (MT-RF (MT-step MT MA sigs))
                  (ISA-RF (INST-post-ISA i))))
  :hints (("goal" :in-theory (enable MT-RF INST-in MT-step)
                :use (:instance MT-RF-MT-step-if-INST-commit-help
                                (trace (MT-trace MT))
                                (spc 0) (smc 0)
                                (ISA (MT-init-ISA MT))))
          ("goal'" :cases ((consp (MT-trace MT)))))))

; This is a landmark lemma.  When commit-inst? is 1, the instruction
; at the head of the ROB commits, advancing the commit boundary
; one instruction ahead.  Thus, MT-RF returns the register file in
; the post-ISA state of the instruction at the head of the ROB.
```

```
;
; Let me explain it more carefully.  MT-RF calculates the
; ideal register file from the MAETT.  Suppose an instruction
; commits in this cycle.  The commit boundary advances one instruction.
; In other words, The instruction specified as
;    (inst-of-tag (MT-ROB-head MT) MT)
; commits.  Thus the register file looks as that in the
; post ISA state of (inst-of-tag (MT-ROB-head MT) MT).
(defthm MT-RF-ISA-RF-inst-of-tag
    (implies (and (inv MT MA)
                  (b1p (commit-inst? MA))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (MT-RF (MT-step MT MA sigs))
                    (ISA-RF
                     (INST-post-ISA (inst-of-tag (MT-ROB-head MT) MT)))))
  :hints (("goal" :restrict ((MT-RF-MT-step-if-INST-commit?
                              ((i (inst-of-tag (MT-ROB-head MT) MT))))))))
)

(encapsulate nil
(local
(defthm MT-RF-INST-pre-ISA-if-MT-all-commit-before-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (member-equal i trace) (INST-p i)
                  (trace-all-commit-before i trace)
                  (subtrace-p trace MT)
                  (not (committed-p i)))
             (equal (ISA-RF (INST-pre-ISA i))
                    (trace-RF trace
                              (ISA-RF (ISA-before trace MT)))))
  :hints (("goal" :in-theory (enable ISA-before-MT-non-nil-trace)))))

(local
(defthm MT-RF-INST-pre-ISA-if-MT-all-commit-before
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (MT-all-commit-before i MT)
                  (not (committed-p i)))
             (equal (ISA-RF (INST-pre-ISA i))
                    (MT-RF MT)))
  :hints (("goal" :in-theory (e/d (MT-RF MT-all-commit-before
                                                 INST-in)
                                  (MT-RF-=-MA-RF))))))

; Suppose the ROB is not empty.
; The register file in the current MA state is the same as in the
; pre-ISA state of the instruction at the head of the ROB.
(defthm MA-RF-pre-ISA-RF-INST-at-ROB-head
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-of-tag (MT-ROB-head MT) MT))
             (equal (ISA-RF (INST-pre-ISA (inst-of-tag (MT-ROB-head MT) MT)))
                    (MA-RF MA))))
)

; If INST-wb-SRF? is 1 for instruction i, it does not modify
; the general register file during its ISA execution.
(defthm ISA-RF-ISA-step-if-INST-wb-sreg
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
```

```
                (INST-p i) (INST-in i MT)
                (b1p (INST-wb-sreg? i)))
           (equal (ISA-RF (ISA-step (INST-pre-ISA i) intr))
                  (ISA-RF (INST-pre-ISA i))))
  :hints (("goal" :in-theory (enable ISA-step-functions ISA-step
                                     INST-wb-sreg? INST-cntlv
                                     INST-opcode
                                     opcode-inst-type))))

; If instruction i raises an exception, it does not modify
; the register file during ISA execution.
(defthm ISA-RF-ISA-step-if-not-INST-excpt
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (b1p (INST-excpt? i)))
             (equal (ISA-RF (ISA-step (INST-pre-ISA i) intr))
                    (ISA-RF (INST-pre-ISA i))))
  :hints (("goal" :in-theory (enable ISA-step-functions ISA-step
                                     INST-excpt?
                                     INST-fetch-error?
                                     INST-decode-error?
                                     INST-data-access-error?
                                     INST-STORE-ERROR?
                                     INST-LOAD-ERROR?
                                     DECODE-ILLEGAL-INST?
                                     INST-RA
                                     lift-b-ops))))

; This is an important lemma.
;  The effect of writeback instruction i.
(defthm ISA-RF-ISA-step-if-INST-wb
     (implies (and (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (INST-p i) (INST-in i MT)
                   (b1p (INST-wb? i))
                   (not (b1p (INST-wb-sreg? i)))
                   (not (b1p (INST-excpt? i)))
                   (not (b1p (ISA-input-exint intr))))
              (equal (ISA-RF (ISA-step (INST-pre-ISA i) intr))
                     (write-reg (INST-dest-val i)
                                (INST-dest-reg i)
                                (ISA-RF (INST-pre-ISA i))))))
  :hints (("goal" :in-theory (enable ISA-step-functions ISA-step
                                     INST-excpt?
                                     INST-fetch-error?
                                     INST-decode-error?
                                     INST-data-access-error?
                                     INST-STORE-ERROR?
                                     INST-LOAD-ERROR?
                                     DECODE-ILLEGAL-INST?
                                     INST-wb?
                                     INST-wb-sreg? INST-cntlv
                                     INST-opcode
                                     opcode-inst-type
                                     INST-DEST-VAL
                                     INST-DEST-REG
                                     INST-ADD-DEST-VAL
                                     INST-MULT-DEST-VAL
                                     INST-LD-DEST-VAL
                                     INST-LD-IM-DEST-VAL
                                     INST-MFSR-DEST-VAL
```

```
                                    INST-MTSR-DEST-VAL
                                    INST-RA INST-RB INST-RC
                                    INST-IM
                                    INST-SRC-VAL1
                                    INST-SRC-VAL2
                                    lift-b-ops))))

; Suppose instruction i is at the head of the ROB.
; If exception handling starts in this cycle, then i must be an exception-
; raising instruction.
(defthm INST-excpt-INST-at-ROB-head-if-enter-excpt
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-of-tag (MT-rob-head MT) MT)
                  (not (b1p (INST-modified? (inst-of-tag (MT-rob-head MT)
                                                         MT))))
                  (not (b1p (inst-specultv? (inst-of-tag (MT-rob-head MT)
                                                         MT))))
                  (b1p (enter-excpt? MA)))
             (equal (INST-excpt? (inst-of-tag (MT-ROB-head MT) MT)) 1))
  :hints (("goal" :in-theory (enable enter-excpt? lift-b-ops
                                     equal-b1p-converter))))

; An important lemma.
; Output ROB-write-val from the ROB is equal to the destination value
; of the instruction at the head of the ROB.
(defthm ROB-write-val-INST-dest-val
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-of-tag (MT-ROB-head MT) MT)
                  (complete-stg-p
                   (INST-stg (inst-of-tag (MT-ROB-head MT) MT)))
                  (INST-writeback-p (inst-of-tag (MT-ROB-head MT) MT))
                  (not (b1p (INST-modified?
                             (inst-of-tag (MT-ROB-head MT) MT))))
                  (not (b1p (inst-specultv?
                             (inst-of-tag (MT-ROB-head MT) MT))))
                  (not (b1p (INST-excpt?
                             (inst-of-tag (MT-ROB-head MT) MT)))))
             (equal (ROB-write-val (MA-rob MA) MA)
                    (INST-dest-val (inst-of-tag (MT-ROB-head MT) MT))))
  :hints (("goal" :in-theory (enable ROB-write-val))))

; An important lemma.
; Output of ROB-write-rid from the ROB designates the destination register
; of the instruction at the head of the ROB.
(defthm ROB-write-rid-INST-dest-reg
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-of-tag (MT-ROB-head MT) MT)
                  (complete-stg-p
                   (INST-stg (inst-of-tag (MT-ROB-head MT) MT)))
                  (INST-writeback-p (inst-of-tag (MT-ROB-head MT) MT))
                  (not (b1p (INST-modified?
                             (inst-of-tag (MT-ROB-head MT) MT))))
                  (not (b1p (inst-specultv?
                             (inst-of-tag (MT-ROB-head MT) MT))))
                  (not (b1p (INST-excpt?
                             (inst-of-tag (MT-ROB-head MT) MT)))))
             (equal (ROB-write-rid (MA-rob MA))
                    (INST-dest-reg (inst-of-tag (MT-ROB-head MT) MT))))
  :hints (("goal" :in-theory (enable ROB-write-rid exception-relations))))
```

```
; If instruction i is not a write-back instruction, it does not modify
; the register file in ISA execution.
(defthm ISA-RF-ISA-step-if-not-INST-wb
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (not (b1p (INST-wb? i))))
             (equal (ISA-RF (ISA-step (INST-pre-ISA i) intr))
                    (ISA-RF (INST-pre-ISA i))))
  :hints (("goal" :in-theory (enable ISA-step-functions ISA-step
                                     INST-wb? INST-cntlv
                                     INST-opcode
                                     opcode-inst-type))))

; An important lemma. One step before
;   step-RF-INST-post-ISA-of-INST-at-MT-ROB-head
; Suppose instruction i is at the head of the ROB.  If i commits in
; this cycle, the register file states in the next MA state is that of
; the post-ISA state of i.
(defthm step-RF-INST-post-ISA-of-INST-at-MT-ROB-head-normal-case
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (commit-inst? MA))
                  (not (b1p (INST-modified?
                             (inst-of-tag (MT-ROB-head MT) MT))))
                  (not (b1p (inst-specultv?
                             (inst-of-tag (MT-ROB-head MT) MT)))))
             (equal (step-RF MA)
                    (ISA-RF
                     (INST-post-ISA (inst-of-tag (MT-ROB-head MT) MT)))))
  :hints (("goal" :in-theory (e/d (step-RF lift-b-ops
                                     INST-exintr-INST-in-if-not-retired
                                     commit-inst? ROB-write-reg?)
                                  (inst-is-at-one-of-the-stages))
                  :cases ((b1p (INST-fetch-error?
                                (inst-of-tag (MT-ROB-head MT) MT)))))
          ("subgoal 2" :cases ((b1p (INST-excpt?
                                     (inst-of-tag (MT-ROB-head MT) MT)))))
          ("subgoal 1" :cases ((b1p (INST-excpt?
                                     (inst-of-tag (MT-ROB-head MT) MT))))
                       :in-theory (enable INST-excpt? lift-b-ops
                                          commit-inst? ROB-write-reg?
                                          exception-relations
                                          step-RF))
          ("subgoal 2.2" :use
                         (:instance INST-is-at-one-of-the-stages
                                    (i (inst-of-tag (MT-ROB-head MT) MT))))
          ("subgoal 2.1" :use
                         (:instance INST-is-at-one-of-the-stages
                                    (i (inst-of-tag (MT-ROB-head MT) MT))))))

; A landmark lemma.
; A corollary of
;  step-RF-INST-post-ISA-of-INST-at-MT-ROB-head-normal-case.
; The general register file in the next cycle is the same as in the
; post-ISA of the instruction at the head of the ROB.
(defthm step-RF-INST-post-ISA-of-INST-at-MT-ROB-head
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (commit-inst? MA))
                  (not (MT-CMI-p (MT-step MT MA sigs))))
```

```
            (equal (step-RF MA)
                   (ISA-RF
                    (INST-post-ISA (inst-of-tag (MT-ROB-head MT) MT)))))
  :hints (("goal" :cases
                  ((b1p (INST-modified? (inst-of-tag (MT-ROB-head MT) MT)))
                   (b1p (inst-specultv? (inst-of-tag (MT-ROB-head MT) MT))))
                  :in-theory (enable commit-inst? lift-b-ops))))

(local
(defthm RF-match-p-preserved-help
    (implies (and (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p intr)
                  (inv MT MA)
                  (not (MT-CMI-p (MT-step MT ma intr))))
             (equal (MT-RF  (MT-step MT ma intr))
                    (step-RF MA)))
  :hints (("goal" :cases ((b1p (commit-inst? MA)))))))


; RF-match-p is preserved.
(defthm RF-match-p-preserved
    (implies (and (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p intr)
                  (inv MT MA)
                  (not (MT-CMI-p (MT-step MT ma intr))))
             (RF-match-p (MT-step MT ma intr) (MA-step ma intr)))
  :hints (("goal" :in-theory (enable RF-match-p
                                     MT-CMI-p))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Proof of SRF-Match
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Initial special register satisfies SRF-match-p
(defthm SRF-match-p-init-MA
    (implies (and (MA-state-p MA) (b1p (MA-flushed? MA)))
             (SRF-match-p (init-MT MA) MA))
  :Hints (("goal" :in-theory (enable SRF-match-p init-MT
                                     MT-SRF))))

;;;;;;; Induction case
;
; There are three cases to consider.
;  1  An external interrupt occurs.
;  2  No exception, no instruction commit occurs.
;  3  Commitment of an instruction occurs.
;  Commitment includes internal exception handling in our case.
;
; There are seven landmark lemmas:
; For Case 1
;     step-SRF-if-ex-intr?
;     MT-SRF-MT-step-if-ex-intr
;     ISA-SRF-ISA-step-exintr
; For Case 2,
;     step-SRF-if-not-commit-inst?
;     MT-SRF-MT-step-if-not-commit-inst
; For Case 3,
;    MT-SRF-ISA-RF-inst-of-tag
;    step-SRF-INST-post-ISA-of-INST-at-MT-ROB-head

;
; We define MT-ISA-at-dispatch to calculate the ISA state at the
; dispatching boundary.  This state can be characterized as the
```

```
; post-ISA of the most recently dispatched instruction, or the pre-ISA
; of the first instruction that has not been dispatched yet.
(defun trace-ISA-at-dispatch (trace ISA)
  (if (endp trace)
      ISA
      (if (not (dispatched-p (car trace)))
          ISA
          (trace-ISA-at-dispatch (cdr trace) (INST-post-ISA (car trace))))))

(defun MT-ISA-at-dispatch (MT)
  (trace-ISA-at-dispatch (MT-trace MT) (MT-init-ISA MT)))

(in-theory (disable MT-ISA-at-dispatch))

;;; Start Case 1
; If no instruction commits in this cycle, the special registers are not
; updated.
(defthm commit-inst-if-rob-write-sreg
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (commit-inst? MA))))
             (equal (ROB-write-sreg? (MA-rob MA))  0))
  :hints (("goal" :in-theory (enable commit-inst? ROB-write-sreg?
                                     lift-b-ops equal-b1p-converter))))

; An landmark lemma.
; The effect of an external interrupt on special register file.
(defthm step-SRF-if-ex-intr?
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (ex-intr? MA sigs)))
             (equal (step-SRF MA sigs)
                    (SRF 1 (word (ex-intr-addr MA))
                           (word (SRF-su (MA-SRF MA))))))
  :hints (("goal" :in-theory (enable step-SRF))))

(encapsulate nil
(local
(defthm commit-inst-p-if-dispatched-and-ex-intr
    (implies (and (inv MT MA)
                  (dispatched-p i)
                  (b1p (EX-intr? MA sigs))
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA))
             (committed-p i))
  :hints (("goal" :in-theory
                  (enable NOT-EX-INTR-IF-INST-AT-EXECUTE-OR-COMPLETE-STG)))))

(local
(defthm MT-SRF-MT-step-if-ex-intr-help
    (implies (and (inv MT MA)
                  (b1p (ex-intr? MA sigs))
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (trace-SRF (step-trace trace MT MA sigs
                                           ISA spc smc)
                               (ISA-SRF ISA))
                    (ISA-SRF
                     (ISA-step (trace-ISA-at-dispatch trace ISA)
                               (ISA-input 1)))))
  :hints (("goal" :in-theory (e/d (trace-SRF*)
```

```
                                  (trace-SRF))))
  :rule-classes nil))

; This is a landmark lemma.  This characterize the ideal state
; of the special register in the next state after an external interrupt
; occurs.
;
; If an external interrupt occurs, the ideal state of the special register
; file is obtained by expanding the definition of the ISA step function
; for an external interrupt.
(defthm MT-SRF-MT-step-if-ex-intr
    (implies (and (inv MT MA)
                  (b1p (ex-intr? MA sigs))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (MT-SRF (MT-step MT MA sigs))
                    (ISA-SRF
                     (ISA-step (MT-ISA-at-dispatch MT)
                               (ISA-input 1)))))
  :hints (("goal" :use (:instance
                        MT-SRF-MT-step-if-ex-intr-help
                        (trace (MT-trace MT)) (smc 0) (spc 0)
                        (ISA  (MT-init-ISA MT)))
                  :in-theory (enable MT-SRF MT-step
                                     MT-ISA-at-dispatch))))
)

(encapsulate nil
(local
(defthm not-MT-all-commit-before-if-INST-modified-help
    (implies (and (member-equal i trace)
                  (not (equal i (car trace)))
                  (b1p (INST-modified? i))
                  (trace-correct-modified-first trace)
                  (not (b1p (INST-first-modified? i)))
                  (not (trace-CMI-p trace)))
             (not (trace-all-commit-before i trace)))))

; A nasty corner case lemma about the relation between
; The reader can skip it, without losing the picture of the proof.
; MT-CMI-p, INST-modified? and INST-first-modified? are involved.
;
; If INST-first-modified? is false, there must be an modified
; instruction before i. Name it j. And if all instruction before i are
; committed, that implies j is committed. Thus MT-CMI-p
; must be non-nil, if all instructions preceding i are committed.
(defthm not-MT-all-commit-before-if-INST-modified
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (b1p (INST-modified? i))
                  (not (b1p (INST-first-modified? i)))
                  (not (MT-CMI-p MT))
                  (MAETT-p MT) (MA-state-p MA))
             (not (MT-all-commit-before i MT)))
  :hints (("goal" :cases ((consp (MT-trace MT))))
          ("subgoal 2" :in-theory (enable INST-in))
          ("subgoal 1" :cases ((equal (car (MT-trace MT)) i)))
          ("subgoal 1.2" :in-theory
                         (enable MT-all-commit-before INST-in inv
                                 correct-modified-first weak-inv
                                 MT-CMI-p))))
)
```

```
; Following lemmas are about the effect of an external interrupt on
; special register.  Many lemmas involve MT-ISA-at-dispatch, which is
; a new function that requires minimum theory build-up.
;
; WARNING!!!!
; Following lemmas may be boring lemmas for a while.
; If you are not interested in the detail, skip to the landmark lemma
; ISA-SRF-ISA-step-exintr, where things get interesting again.
(encapsulate nil
(local
 (defthm commit-if-earlier-than-dq0-and-rob-empty
     (implies (and (inv MT MA)
                   (subtrace-p trace MT)
                   (consp trace)
                   (equal (INST-stg i) '(DQ 0))
                   (member-equal i (cdr trace))
                   (b1p (rob-empty? (MA-rob MA)))
                   (not (commit-stg-p (INST-stg (car trace))))
                   (INST-p i) (INST-listp trace)
                   (MAETT-p MT) (MA-state-p MA))
              (retire-stg-p (INST-stg (car trace))))
   :hints (("goal" :use ((:instance INST-is-at-one-of-the-stages
                                    (i (car trace)))
                         (:instance DQ0-IS-EARLIER-THAN-OTHER-DQ
                                    (i i) (j (car trace))))
                   :in-theory (disable INST-is-at-one-of-the-stages)
                   :restrict ((NOT-ROB-EMPTY-IF-INST-IS-EXECUTED
                               ((i (car trace)))))))))

(local
(defthm MT-all-commit-before-INST-at-DQ0-induct
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (member-equal i trace) (INST-p i)
                  (b1p (rob-empty? (MA-rob MA)))
                  (equal (INST-stg i) '(DQ 0))
                  (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA))
             (trace-all-commit-before i trace))))

(local
(defthm MT-all-commit-before-INST-at-DQ0-help
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(DQ 0))
                  (b1p (rob-empty? (MA-rob MA)))
                  (MAETT-p MT) (MA-state-p MA))
             (MT-all-commit-before i MT))
  :hints (("goal" :in-theory (enable MT-all-commit-before INST-in)))))

; When an external interrupt occurs, all issued instructions have been
; committed.
(defthm MT-all-commit-before-INST-at-DQ0
    (implies (and (inv MT MA)
                  (b1p (ex-intr? MA sigs))
                  (uniq-inst-at-stg '(DQ 0) MT)
                  (MAETT-p MT) (MA-state-p MA))
             (MT-all-commit-before (INST-at-stg '(DQ 0) MT) MT))
  :hints (("goal" :in-theory (enable ex-intr? lift-b-ops))))
)

(local
```

```
(defthm no-DQ-inst-if-not-DQ-DE0-valid
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (not (DQ-stg-p (INST-stg i))))
  :hints (("goal" :in-theory (enable DQ-stg-p)))))
(local (in-theory (disable no-DQ-inst-if-not-DQ-DE0-valid)))

(encapsulate nil
(local
 (defthm commit-if-earlier-than-IFU-and-rob-empty
     (implies (and (inv MT MA)
                   (subtrace-p trace MT)
                   (consp trace)
                   (equal (INST-stg i) '(IFU))
                   (member-equal i (cdr trace))
                   (b1p (rob-empty? (MA-rob MA)))
                   (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))
                   (not (commit-stg-p (INST-stg (car trace))))
                   (INST-p i) (INST-listp trace)
                   (MAETT-p MT) (MA-state-p MA))
              (retire-stg-p (INST-stg (car trace))))
   :hints (("goal" :use ((:instance INST-is-at-one-of-the-stages
                                    (i (car trace)))
                         )
                   :in-theory (e/d (no-DQ-inst-if-not-DQ-DE0-valid)
                                   (inst-is-at-one-of-the-stages))
                   :restrict ((NOT-ROB-EMPTY-IF-INST-IS-EXECUTED
                               ((i (car trace)))))))))

(local
(defthm MT-all-commit-before-INST-at-IFU-induct
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (b1p (rob-empty? (MA-ROB MA)))
                  (member-equal i trace) (INST-p i) (INST-listp trace)
                  (equal (INST-stg i) '(IFU))
                  (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (trace-all-commit-before i trace))))

(local
(defthm MT-all-commit-before-INST-at-IFU-help
    (implies (and (inv MT MA)
                  (b1p (rob-empty? (MA-ROB MA)))
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(IFU))
                  (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (MT-all-commit-before i MT))
  :hints (("goal" :in-theory (enable MT-all-commit-before INST-in)))))

; If no instruction is in the dispatch queue, and if an external
; interrupt occurs, then all instructions preceding the instruction in
; the IFU are committed.
(defthm MT-all-commit-before-INST-at-IFU
    (implies (and (inv MT MA)
                  (b1p (ex-intr? MA sigs))
                  (uniq-inst-at-stg '(IFU) MT)
                  (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))
                  (MAETT-p MT) (MA-state-p MA))
```

```
            (MT-all-commit-before (INST-at-stg '(IFU) MT) MT))
  :hints (("goal" :in-theory (enable ex-intr? lift-b-ops))))
)

(encapsulate nil
(defthm MT-ISA-at-dispatch-inst-at-DQ0-induct
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (member-equal i trace) (INST-p i) (INST-listp trace)
                  (not (dispatched-p i))
                  (trace-all-commit-before i trace))
             (equal (trace-ISA-at-dispatch trace (INST-pre-ISA (car trace)))
                    (INST-pre-ISA i)))
  :hints ((when-found (car (cdr trace))
                      (:cases ((consp (cdr trace))))))
  :rule-classes nil)

(local
(defthm MT-ISA-at-dispatch-inst-at-DQ0-help
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (dispatched-p i))
                  (MT-all-commit-before i MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (MT-ISA-at-dispatch MT)
                    (INST-pre-ISA i)))
  :hints (("goal" :in-theory (enable MT-ISA-at-dispatch MT-all-commit-before
                                     INST-in)
                  :use (:instance MT-ISA-at-dispatch-inst-at-DQ0-induct
                                  (trace (MT-trace MT))))
          ("goal'" :cases ((consp (MT-trace MT)))))))

; A lemma about MT-ISA-at-dispatch.
; The ISA state at the dispatching boundary is equal to the pre-ISA of DQ0.
(defthm MT-ISA-at-dispatch-inst-at-DQ0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg '(DQ 0) MT)
                  (MT-all-commit-before (INST-at-stg '(DQ 0) MT) MT))
             (equal (MT-ISA-at-dispatch MT)
                    (INST-pre-ISA (INST-at-stg '(DQ 0) MT))))
  :hints (("goal" :restrict
                  ((MT-ISA-at-dispatch-inst-at-DQ0-help
                    ((i (INST-at-stg '(DQ 0) MT)))))))

; A lemma about MT-ISA-at-dispatch.
; If there is no instruction in the dispatch queue, then the ISA state at
; the dispatching boundary is equal to the pre-ISA of the instruction at
; IFU.
(defthm MT-ISA-at-dispatch-inst-at-IFU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg '(IFU) MT)
                  (MT-all-commit-before (INST-at-stg '(IFU) MT) MT))
             (equal (MT-ISA-at-dispatch MT)
                    (INST-pre-ISA (INST-at-stg '(IFU) MT))))
  :hints (("goal" :restrict
                  ((MT-ISA-at-dispatch-inst-at-DQ0-help
                    ((i (INST-at-stg '(IFU) MT)))))))

)
```

```
(local
(encapsulate nil
(local
(defthm ISA-pc-MT-ISA-at-dispatch-MT-pc-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (not (b1p (IFU-valid? (MA-IFU MA))))
                  (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))
                  (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (ISA-PC (trace-ISA-at-dispatch trace ISA))
                    (trace-pc trace (ISA-pc ISA))))
  :Hints (("goal" :in-theory (e/d (dispatched-p* dq-stg-p)
                                  (dispatched-p))
                  :restrict ((DQ-DE0-valid-if-inst-in ((I (car trace))))
                             (DQ-DE1-valid-if-inst-in ((I (car trace))))
                             (DQ-DE2-valid-if-inst-in ((I (car trace))))
                             (DQ-DE3-valid-if-inst-in ((I (car trace)))))))))

; A help lemma to prove ISA-pc-MT-ISA-at-dispatch
;
; If no instruction is at IFU or dispatch queue, then the ISA at the
; dispatching boundary has the same pc value as the PC in the
; actual MA state.
;
; Note: we need to prove about PC here, because the special register
; stores the correct PC value when an exception occurs.
(defthm ISA-pc-MT-ISA-at-dispatch-MT-pc
    (implies (and (inv MT MA)
                  (not (b1p (IFU-valid? (MA-IFU MA))))
                  (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (ISA-PC (MT-ISA-at-dispatch MT))
                    (MT-pc MT)))
  :hints (("goal" :in-theory (enable MT-pc MT-ISA-at-dispatch))))
))

(local
(encapsulate nil
(local
 (defthm committed-p-if-not-IFU-DE-valid-rob-empty
     (implies (and (inv MT MA)
                   (INST-in i MT) (INST-p i)
                   (b1p (rob-empty? (MA-ROB MA)))
                   (not (b1p (IFU-valid? (MA-IFU MA))))
                   (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))
                   (not (commit-stg-p (INST-stg i)))
                   (MAETT-p MT) (MA-state-p MA))
              (retire-stg-p (INST-stg i)))
   :hints (("goal" :use (:instance INST-is-at-one-of-the-stages)
                   :in-theory (e/d (DQ-stg-p)
                                   (inst-is-at-one-of-the-stages))))))

(local
 (defthm not-inst-specultv-help
     (implies (and (inv MT MA)
                   (INST-in i MT) (INST-p i)
                   (b1p (rob-empty? (MA-ROB MA)))
                   (not (b1p (IFU-valid? (MA-IFU MA))))
                   (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))
                   (MAETT-p MT) (MA-state-p MA))
```

```
            (equal (inst-specultv? i) 0))
   :hints (("goal" :use (:instance
                         NOT-INST-SPECULTV-INST-IN-IF-COMMITTED)
                   :in-theory (enable committed-p)))))



(local
(defthm MT-specultv-if-not-IFU-DE-valid-rob-empty-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (b1p (rob-empty? (MA-ROB MA)))
                  (not (b1p (IFU-valid? (MA-IFU MA))))
                  (not (b1p (DE-valid? (DQ-DEO (MA-DQ MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (trace-specultv? trace) 0))
  :hints (("goal" :in-theory (enable lift-b-ops INST-start-specultv?
                                     committed-p)))))

(local
(defthm MT-specultv-if-not-IFU-DE-valid-rob-empty
    (implies (and (inv MT MA)
                  (b1p (rob-empty? (MA-ROB MA)))
                  (not (b1p (IFU-valid? (MA-IFU MA))))
                  (not (b1p (DE-valid? (DQ-DEO (MA-DQ MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (MT-specultv? MT) 0))
  :hints (("goal" :in-theory (enable MT-specultv?)))))

(local
 (defthm MT-self-modify-if-not-IFU-DE-valid-rob-empty-induct
     (implies (and (inv MT MA)
                   (subtrace-p trace MT) (INST-listp trace)
                   (not (trace-CMI-p trace))
                   (b1p (rob-empty? (MA-ROB MA)))
                   (not (b1p (IFU-valid? (MA-IFU MA))))
                   (not (b1p (DE-valid? (DQ-DEO (MA-DQ MA)))))
                   (MAETT-p MT) (MA-state-p MA))
              (equal (trace-self-modify? trace) 0))
   :hints (("goal" :in-theory (enable committed-p)))))

(local
 (defthm MT-self-modify-if-not-IFU-DE-valid-rob-empty
     (implies (and (inv MT MA)
                   (not (MT-CMI-p MT))
                   (b1p (rob-empty? (MA-ROB MA)))
                   (not (b1p (IFU-valid? (MA-IFU MA))))
                   (not (b1p (DE-valid? (DQ-DEO (MA-DQ MA)))))
                   (MAETT-p MT) (MA-state-p MA))
              (equal (MT-self-modify? MT) 0))
   :hints (("goal" :in-theory (enable MT-self-modify? MT-CMI-p)))))

; Lemma to help the proof of ISA-pc-MT-ISA-at-dispatch.
(defthm ISA-pc-MT-ISA-at-dispatch-if-not-IFU-DE-valid
    (implies (and (inv MT MA)
                  (b1p (ex-intr? MA sigs))
                  (not (MT-CMI-p MT))
                  (not (b1p (IFU-valid? (MA-IFU MA))))
                  (not (b1p (DE-valid? (DQ-DEO (MA-DQ MA)))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (ISA-PC (MT-ISA-at-dispatch MT))
```

```
                  (MA-pc MA)))
  :hints (("goal" :in-theory (enable ex-intr? lift-b-ops))))
))

(local (in-theory (disable ISA-pc-MT-ISA-at-dispatch-MT-pc)))

(encapsulate nil
(local
(defthm ISA-pc-MT-ISA-at-dispatch-help1
    (implies (and (inv MT MA)
                  (b1p (ex-intr? MA sigs))
                  (not (MT-CMI-p MT))
                  (uniq-inst-at-stg '(DQ 0) MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (DE-PC (DQ-DE0 (MA-DQ MA)))
                  (INST-pc (INST-at-stg '(DQ 0) MT))))
  :hints (("goal" :cases ((MT-all-commit-before (INST-at-stg '(DQ 0) MT) MT)))
          ("subgoal 1" :in-theory (disable MT-ALL-COMMIT-BEFORE-INST-AT-DQ0)
               :cases ((b1p (inst-specultv?
                             (INST-at-stg '(DQ 0) MT)))
                       (and (b1p (INST-modified?
                                  (INST-at-stg '(DQ 0) MT)))
                            (not (b1p (INST-first-modified?
                                       (INST-at-stg '(DQ 0) MT))))))))))

(local
(defthm ISA-pc-MT-ISA-at-dispatch-help2
    (implies (and (inv MT MA)
                  (b1p (ex-intr? MA sigs))
                  (not (MT-CMI-p MT))
                  (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))
                  (uniq-inst-at-stg '(IFU) MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (IFU-PC (MA-IFU MA))
                  (INST-pc (INST-at-stg '(IFU) MT))))
  :hints (("goal" :cases ((MT-all-commit-before (INST-at-stg '(IFU) MT) MT)))
          ("subgoal 1" :in-theory (disable MT-ALL-COMMIT-BEFORE-INST-AT-IFU)
                       :cases ((b1p (inst-specultv?
                                     (INST-at-stg '(IFU) MT)))
                               (and (b1p (INST-modified?
                                          (INST-at-stg '(IFU) MT)))
                                    (not (b1p (INST-first-modified?
                                               (INST-at-stg '(IFU) MT))))))))))

; If external interrupt occurs, the pc value in the ISA at the dispatching
; boundary is output from ex-intr-addr.
(defthm ISA-pc-MT-ISA-at-dispatch
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (ex-intr? MA sigs))
                  (not (MT-CMI-p MT)))
             (equal (ISA-pc (MT-ISA-at-dispatch MT))
                  (ex-intr-addr MA)))
  :hints (("goal" :in-theory (enable ex-intr-addr))))
)

(encapsulate  nil
(local
(defthm ISA-SRF-su-MT-ISA-at-dispatch-induction
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
```

```
                (b1p (rob-empty? (MA-rob MA)))
                (MAETT-p MT) (MA-state-p MA))
           (equal (SRF-su (ISA-SRF (trace-ISA-at-dispatch trace ISA)))
                  (SRF-su (trace-SRF trace (ISA-SRF ISA)))))
  :hints (("goal" :restrict ((NOT-ROB-EMPTY-IF-INST-IS-EXECUTED
                              ((i (car trace)))))))))

(local
(defthm ISA-SRF-su-MT-ISA-at-dispatch-help
    (implies (and (inv MT MA)
                (b1p (rob-empty? (MA-rob MA)))
                (MAETT-p MT) (MA-state-p MA))
           (equal (SRF-su (ISA-SRF (MT-ISA-at-dispatch MT)))
                  (SRF-su (MT-SRF MT))))
  :hints (("goal" :in-theory (e/d (MT-ISA-at-dispatch MT-SRF)
                                  (MT-SRF-=-MA-SRF))))))

; The current supervisor/user mode is the same as in the ISA at the
; dispatching boundary.
(defthm ISA-SRF-su-MT-ISA-at-dispatch
    (implies (and (inv MT MA)
                (b1p (ex-intr? MA MA-sigs))
                (MAETT-p MT) (MA-state-p MA))
           (equal (SRF-su (ISA-SRF (MT-ISA-at-dispatch MT)))
                  (SRF-su (MA-SRF MA))))
  :Hints (("goal" :in-theory (enable ex-intr? lift-b-ops))))
)

; This is a landmark lemma.
; The effect of an external interrupt on the special register file.
(defthm ISA-SRF-ISA-step-exintr
    (implies (and (inv MT MA) (b1p (ISA-input-exint ISA-sigs))
                (MAETT-p MT) (MA-state-p MA)
                (MA-input-p MA-sigs)
                (b1p (ex-intr? MA MA-sigs))
                (not (MT-CMI-p (MT-step MT MA MA-sigs))))
           (equal (ISA-SRF (ISA-step (MT-ISA-at-dispatch MT) ISA-sigs))
                  (SRF 1 (word (ex-intr-addr MA))
                         (word (SRF-su (MA-SRF MA))))))
  :hints (("goal" :in-theory (enable ISA-step ISA-external-intr)
                  :cases ((MT-CMI-p MT)))))

;;; Starting Case 2

; A landmark lemma, even though it is easy to prove.
; If no instruction commits and no external interrupt occurs, then
; the special register does not change.
(defthm step-SRF-if-not-commit-inst?
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (ex-intr? MA sigs)))
                (not (b1p (commit-inst? MA))))
           (equal (step-SRF MA sigs) (MA-SRF MA)))
  :hints (("goal" :in-theory (enable step-SRF))))

(encapsulate nil
(local
(defthm MT-SRF-MT-step-if-not-commit-inst-help
    (implies (and (inv MT MA)
                (not (b1p (commit-inst? MA)))
                (not (b1p (ex-intr? MA sigs))))
           (equal (trace-SRF (step-trace trace MT MA sigs
```

```
                                        ISA spc smc)
                              (ISA-SRF ISA))
                    (trace-SRF trace (ISA-SRF ISA))))))

; This is a landmark lemma.
; If no instruction commits this cycle, MT-SRF remains identical.
(defthm MT-SRF-MT-step-if-not-commit-inst
    (implies (and (inv MT MA)
                  (not (b1p (ex-intr? MA sigs)))
                  (not (b1p (commit-inst? MA))))
             (equal (MT-SRF (MT-step MT MA sigs))
                    (MT-SRF MT)))
  :hints (("goal" :in-theory (enable MT-SRF MT-step))))
)

;;; Start Case 3
(encapsulate nil
(local
 (defthm INST-commit-ex-intr-exclusive
     (implies (and (inv MT MA)
                   (MA-state-p MA) (MAETT-p MT)
                   (b1p (ex-intr? MA sigs)))
              (not (b1p (INST-commit? i MA))))
   :Hints (("goal" :in-theory (enable INST-commit? lift-b-ops)))))

(local
(defthm MT-SRF-MT-step-if-INST-commit-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (member-equal i trace)
                  (b1p (INST-commit? i MA))
                  (INST-p i) (MAETT-p MT) (MA-state-p MA))
             (equal (trace-SRF (step-trace trace MT MA sigs
                                        ISA spc smc)
                               (ISA-SRF (INST-pre-ISA (car trace))))
                    (ISA-SRF (INST-post-ISA i))))
  :rule-classes nil
  :hints (("goal" :in-theory (e/d (trace-SRF*
                                   INST-exintr-now-INST-commit-exclusive)
                                  (trace-SRF))
                  :induct t)
          (when-found (INST-pre-ISA (car (cdr trace)))
                      (:cases ((consp (cdr trace)))))
          (when-found (consp (cdr trace))
                      (:expand
                       (STEP-TRACE (CDR TRACE)
                           MT MA SIGS
                           (ISA-STEP (INST-PRE-ISA (CAR TRACE))
                                     (ISA-input (INST-EXINTR? (CAR TRACE))))
                           (B-IOR (INST-SPECULTV? (CAR TRACE))
                                  (INST-START-SPECULTV? (CAR TRACE)))
                           (INST-MODIFIED? (CAR TRACE))))))))

(local
(defthm MT-SRF-MT-step-if-INST-commit?
    (implies (and (inv MT MA)
                  (b1p (INST-commit? i MA))
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (MT-SRF (MT-step MT MA sigs))
                    (ISA-SRF (INST-post-ISA i))))
```

```
  :hints (("goal" :in-theory (enable MT-SRF INST-in MT-step)
                  :use (:instance MT-SRF-MT-step-if-INST-commit-help
                                  (trace (MT-trace MT))
                                  (spc 0) (smc 0)
                                  (ISA (MT-init-ISA MT))))
          ("goal'" :cases ((consp (MT-trace MT))))))

; This is another landmark lemma.
; If a instruction commits this cycle, the special register file in the
; next cycle is the same as in the post-ISA state of the instruction
; at the head of the ROB.
;
; Let me explain it more carefully. commit-inst? = 1 suggests that
; the instruction at the head of the ROB is committing this cycle.
; This advances the committing boundary one instruction ahead.
; MT-SRF in the next cycle returns the special register
; file of the post-ISA state of the instruction, because the
; new commitment boundary is after the instruction.
(defthm MT-SRF-ISA-RF-inst-of-tag
    (implies (and (inv MT MA)
                  (b1p (commit-inst? MA))
                  (not (b1p (ex-intr? MA sigs)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (MT-SRF (MT-step MT MA sigs))
                    (ISA-SRF
                     (INST-post-ISA (inst-of-tag (MT-ROB-head MT) MT)))))
  :hints (("goal" :restrict ((MT-SRF-MT-step-if-INST-commit?
                              ((i (inst-of-tag (MT-ROB-head MT) MT)))))
                  :in-theory (enable commit-inst? INST-commit?
                                     lift-b-ops))))
)

; From here we start proving the landmark lemma
;     step-SRF-INST-post-ISA-of-INST-at-MT-ROB-head.
; WARNING !! Some lemmas may be boring.
(encapsulate nil
(local
(defthm MT-SRF-INST-pre-ISA-if-MT-all-commit-before-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (member-equal i trace) (INST-p i)
                  (trace-all-commit-before i trace)
                  (subtrace-p trace MT)
                  (not (committed-p i)))
             (equal (ISA-SRF (INST-pre-ISA i))
                    (trace-SRF trace
                               (ISA-SRF (ISA-before trace MT)))))
  :hints (("goal" :in-theory (enable ISA-before-MT-non-nil-trace)))))

(local
(defthm MT-SRF-INST-pre-ISA-if-MT-all-commit-before
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (MT-all-commit-before i MT)
                  (not (committed-p i)))
             (equal (ISA-SRF (INST-pre-ISA i))
                    (MT-SRF MT)))
  :hints (("goal" :in-theory (e/d (MT-SRF MT-all-commit-before
                                                  INST-in)
                                  (MT-SRF-=-MA-SRF))))))
```

```
; Suppose i is an instruction at the head of the ROB.  The current
; state of the special register file is the same as that in the
; pre-ISA state of i.
(defthm MA-SRF-pre-ISA-SRF-INST-at-ROB-head
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-of-tag (MT-ROB-head MT) MT))
             (equal (ISA-SRF (INST-pre-ISA (inst-of-tag (MT-ROB-head MT)
                                                        MT)))
                    (MA-SRF MA))))
)

(encapsulate nil
(local
 (defthm not-INST-excpt-INST-at-ROB-head-if-not-enter-excpt-help
     (implies (and (MA-state-p MA)
                   (not (b1p (enter-excpt? MA)))
                   (b1p (commit-inst? MA)))
              (not (b1p (excpt-raised? (robe-excpt
                                        (nth-robe (rob-head (MA-rob MA))
                                                  (MA-rob MA)))))))
   :hints (("goal" :in-theory (enable lift-b-ops commit-inst? enter-excpt?)))))

; If the exception handling does not start this cycle, the instruction
; at the head of the ROB is not an exception causing instruction.
(defthm not-INST-excpt-INST-at-ROB-head-if-not-enter-excpt
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (INST-modified? (inst-of-tag (MT-rob-head MT)
                                                         MT))))
                  (not (b1p (enter-excpt? MA)))
                  (b1p (commit-inst? MA)))
             (equal (INST-excpt? (inst-of-tag (MT-ROB-head MT) MT)) 0))
  :hints (("goal" :in-theory (enable equal-b1p-converter lift-b-ops)
                  :use
                  (:instance
                   not-INST-excpt-INST-at-ROB-head-if-not-enter-excpt-help))))

)

; The instruction at the head of the ROB is not a RFEH instruction
; if leave-excpt is 0.
(defthm not-INST-rfeh-INST-at-ROB-head-if-not-leave-excpt
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (INST-modified? (inst-of-tag (MT-rob-head MT)
                                                         MT))))
                  (not (b1p (INST-fetch-error? (inst-of-tag (MT-rob-head MT)
                                                            MT))))
                  (not (b1p (leave-excpt? MA)))
                  (b1p (commit-inst? MA)))
             (equal (INST-rfeh? (inst-of-tag (MT-ROB-head MT) MT)) 0))
  :hints (("goal" :in-theory (enable equal-b1p-converter lift-b-ops
                                     leave-excpt? commit-inst?))))

; The instruction at the head of the ROB is a RFEH instruction if
; leave-excpt is 1.
(defthm INST-rfeh-inst-of-tag
    (implies (and (inv MT MA)
                  (b1p (leave-excpt? MA))
                  (not (b1p (INST-modified?
                             (inst-of-tag (MT-ROB-head MT) MT))))
```

```
                (not (b1p (INST-fetch-error?
                           (inst-of-tag (MT-ROB-head MT) MT))))
                (MAETT-p MT) (MA-state-p MA))
           (equal (INST-rfeh? (inst-of-tag (MT-ROB-head MT) MT)) 1))
  :hints (("goal" :in-theory (enable leave-excpt? lift-b-ops
                                     equal-b1p-converter))))

; If If rob-write-sreg from the ROB is 1, the instruction at the head
; of the ROB is a write-back register.
(defthm INST-wb-inst-of-tag-if-rob-write-sreg
    (implies (and (inv MT MA)
                  (b1p (rob-write-sreg? (MA-rob MA)))
                  (not (b1p (INST-modified?
                           (inst-of-tag (MT-ROB-head MT) MT))))
                  (not (b1p (INST-fetch-error?
                           (inst-of-tag (MT-ROB-head MT) MT))))
                  (MAETT-p MT) (MA-state-p MA))
           (equal (INST-wb? (inst-of-tag (MT-ROB-head MT) MT)) 1))
  :hints (("goal" :in-theory (enable rob-write-sreg? lift-b-ops
                                     equal-b1p-converter))))

; If rob-write-sreg from the ROB is 1, the instruction at the
; head of the ROB is a special register modifier.
(defthm INST-wb-sreg-inst-of-tag-if-rob-write-sreg
    (implies (and (inv MT MA)
                  (b1p (rob-write-sreg? (MA-rob MA)))
                  (not (b1p (INST-modified?
                           (inst-of-tag (MT-ROB-head MT) MT))))
                  (not (b1p (INST-fetch-error?
                           (inst-of-tag (MT-ROB-head MT) MT))))
                  (MAETT-p MT) (MA-state-p MA))
           (equal (INST-wb-sreg? (inst-of-tag (MT-ROB-head MT) MT)) 1))
  :hints (("goal" :in-theory (enable rob-write-sreg? lift-b-ops
                                     equal-b1p-converter))))

; Relation between the signal ROB-write-sreg? from the ROB and the
; control vector of the instruction at the head of the ROB.
(defthm ISA-SRF-ISA-step-if-not-ROB-write-SRF-help
    (implies (and (inv MT MA)
                  (uniq-inst-of-tag (MT-ROB-head MT) MT)
                  (complete-stg-p (INST-stg (inst-of-tag (MT-ROB-head MT) MT)))
                  (not (b1p (rob-write-sreg? (MA-rob MA))))
                  (not (b1p (INST-modified?
                           (inst-of-tag (MT-ROB-head MT) MT))))
                  (not (b1p (INST-excpt?
                           (inst-of-tag (MT-ROB-head MT) MT))))
                  (MAETT-p MT) (MA-state-p MA))
             (or (not (b1p (INST-wb-sreg? (inst-of-tag (MT-ROB-head MT) MT))))
                 (not (b1p (INST-wb? (inst-of-tag (MT-ROB-head MT) MT))))))
  :hints (("goal" :in-theory (enable rob-write-sreg? lift-b-ops
                                     INST-excpt?
                                     equal-b1p-converter)))
  :rule-classes nil)

(local
(defthm ISA-SRF-ISA-step-if-INST-rfeh
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (b1p (INST-rfeh? i))
                  (not (b1p (INST-excpt? i)))
                  (not (b1p (ISA-input-exint intr )))
                  (not (b1p (INST-modified? i)))
```

```
                (not (b1p (inst-specultv? i)))
                (MAETT-p MT) (MA-state-p MA))
           (equal (ISA-SRF (ISA-step (INST-pre-ISA i) intr))
                  (SRF (logcar (SRF-SR1 (ISA-SRF (INST-pre-ISA i))))
                       (SRF-SR0 (ISA-SRF (INST-pre-ISA i)))
                       (SRF-SR1 (ISA-SRF (INST-pre-ISA i))))))
  :hints (("goal" :in-theory (enable ISA-step
                                     INST-rfeh? decode INST-opcode
                                     ISA-RFEH
                                     supervisor-mode?
                                     decode-illegal-inst?
                                     ISA-illegal-inst
                                     ISA-fetch-error
                                     INST-decode-error?
                                     lift-b-ops read-sreg
                                     ISA-step-INST-pre-ISA
                                     INST-excpt?
                                     logbit* rdb INST-cntlv
                                     exception-relations)
                  :cases ((b1p (INST-fetch-error? i)))))))

(local
(defthm ISA-SRF-ISA-step-if-INST-fetch-error
    (implies (and (INST-p i)
                  (b1p (INST-fetch-error? i))
                  (not (b1p (ISA-input-exint intr))))
             (equal (ISA-SRF (ISA-step (INST-pre-ISA i) intr))
                    (SRF 1 (word (INST-pc i))
                           (word (SRF-su (ISA-SRF (INST-pre-ISA i)))))))
  :hints (("goal" :in-theory (enable ISA-step INST-fetch-error?
                                     ISA-fetch-error)))))

(local
(defthm ISA-SRF-ISA-step-if-INST-decode-error
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (INST-fetch-error? i)))
                  (b1p (INST-decode-error? i))
                  (not (b1p (ISA-input-exint intr))))
             (equal (ISA-SRF (ISA-step (INST-pre-ISA i) intr))
                    (SRF 1 (word (+ 1 (INST-pc i)))
                           (word (SRF-su (ISA-SRF (INST-pre-ISA i)))))))
  :hints (("goal" :in-theory (enable ISA-step INST-fetch-error?
                                     ISA-RFEH supervisor-mode?
                                     lift-b-ops decode-illegal-inst?
                                     ISA-MFSR ISA-MTSR INST-RA INST-RC
                                     INST-RB
                                     ISA-illegal-inst
                                     INST-decode-error?)))))

(local
(defthm ISA-SRF-ISA-step-if-INST-data-accs-error
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (INST-fetch-error? i)))
                  (b1p (INST-data-access-error? i))
                  (not (b1p (ISA-input-exint intr))))
             (equal (ISA-SRF (ISA-step (INST-pre-ISA i) intr))
                    (SRF 1 (word (INST-pc i))
                           (word (SRF-su (ISA-SRF (INST-pre-ISA i)))))))
```

```
  :hints (("goal" :in-theory (enable INST-data-access-error? ISA-step
                                     INST-fetch-error? INST-load-error?
                                     INST-store-error? ISA-ldi
                                     ISA-ld ISA-sti ISA-st
                                     ISA-data-accs-error
                                     lift-b-ops)))))
(local
(defthm ISA-SRF-ISA-step-if-INST-excpt
    (implies (and (inv MT MA)
                  (b1p (enter-excpt? MA))
                  (not (b1p (ISA-input-exint intr)))
                  (not (b1p (INST-modified?
                             (inst-of-tag (MT-ROB-head MT) MT))))
                  (not (b1p (inst-specultv?
                             (inst-of-tag (MT-ROB-head MT) MT))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (ISA-SRF
                     (ISA-step (INST-pre-ISA (inst-of-tag (MT-ROB-head MT)
                                                          MT))
                               intr))
                    (SRF 1 (rob-write-val (MA-ROB MA) MA)
                         (word (SRF-su (MA-SRF MA))))))
  :hints (("goal" :in-theory (enable enter-excpt? lift-b-ops
                                     rob-write-val INST-excpt-flags
                                     exception-relations
                                     INST-EXCPT-DETECTED-P)
                  :cases ((not (b1p (INST-fetch-error?
                                     (inst-of-tag (MT-ROB-head MT) MT))))))
          ("subgoal 1" :cases ((not (b1p (INST-decode-error?
                                     (inst-of-tag (MT-ROB-head MT) MT))))))
          ("subgoal 1.1" :cases ((not (b1p (INST-data-access-error?
                                        (inst-of-tag (MT-ROB-head MT) MT)))))))))

; The effect of the ISA execution on the special register file
; when the instruction actually modifies the special register.
(defthm ISA-SRF-ISA-step-if-INST-wb
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (b1p (INST-wb? i))
                  (b1p (INST-wb-sreg? i))
                  (not (b1p (INST-rfeh? i)))
                  (not (b1p (INST-excpt? i)))
                  (not (b1p (INST-modified? i)))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (ISA-input-exint intr )))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (ISA-SRF (ISA-step (INST-pre-ISA i) intr))
                    (write-sreg (INST-dest-val i)
                                (INST-dest-reg i)
                                (ISA-SRF (INST-pre-ISA i)))))
  :hints (("goal" :in-theory (enable ISA-step INST-excpt? lift-b-ops
                                     INST-fetch-error? INST-cntlv
                                     decode logbit* rdb INST-opcode
                                     supervisor-mode? decode-illegal-inst?
                                     INST-dest-val INST-dest-reg
                                     INST-rc INST-ra
                                     INST-MTSR-DEST-VAL INST-src-val1
                                     ISA-MTSR INST-decode-error?
                                     INST-wb? INST-wb-sreg?))))

; When the committed instruction does not modify the special register,
; the state of the special register does not change.
```

```
(defthm ISA-SRF-ISA-step-if-not-INST-wb-INST-wb-SRF
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (b1p (INST-excpt? i)))
                  (not (b1p (INST-rfeh? i)))
                  (or (not (b1p (INST-wb? i)))
                      (not (b1p (INST-wb-sreg? i))))
                  (not (b1p (ISA-input-exint intr )))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (ISA-SRF (ISA-step (INST-pre-ISA i) intr))
                    (ISA-SRF (INST-pre-ISA i))))
  :hints (("goal" :in-theory  (enable ISA-step INST-excpt? lift-b-ops
                                      INST-fetch-error? INST-cntlv
                                      INST-rfeh?
                                      decode logbit* rdb INST-opcode
                                      supervisor-mode? decode-illegal-inst?
                                      INST-dest-val INST-dest-reg
                                      INST-rc INST-ra
                                      INST-MTSR-DEST-VAL INST-src-val1
                                      ISA-add  ISA-mul ISA-br
                                      ISA-ld ISA-ldi ISA-st
                                      ISA-sync ISA-mfsr
                                      ISA-sti
                                      INST-decode-error?
                                      INST-DATA-ACCESS-ERROR?
                                      INST-load-error? INST-store-error?
                                      INST-wb? INST-wb-sreg?))))

(encapsulate nil
(local
(defthm step-SRF-INST-post-ISA-of-INST-at-MT-ROB-head-normal-case-1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (commit-inst? MA))
                  (b1p (INST-fetch-error?
                        (inst-of-tag (MT-ROB-head MT) MT)))
                  (not (b1p (INST-modified?
                             (inst-of-tag (MT-ROB-head MT) MT))))
                  (not (b1p (inst-specultv?
                             (inst-of-tag (MT-ROB-head MT) MT)))))
             (equal (step-SRF MA sigs)
                    (ISA-SRF
                     (INST-post-ISA (inst-of-tag (MT-ROB-head MT) MT)))))
  :hints (("goal" :in-theory (e/d (step-SRF lift-b-ops
                                    leave-excpt? enter-excpt?
                                    INST-exintr-INST-in-if-not-retired
                                    COMMIT-INST?
                                    INST-EXCPT-FLAGS
                                    ROB-WRITE-VAL)
                                  (UNIQ-INST-OF-TAG-IF-COMMIT-INST
                                   INST-is-at-one-of-the-stages
                                   UNIQ-INST-OF-TAG-IF-context-sync))
                  :use (:instance inst-is-at-one-of-the-stages
                                  (i (inst-of-tag (MT-rob-head MT) MT)))))))

(local
(defthm step-SRF-INST-post-ISA-of-INST-at-MT-ROB-head-normal-case-2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (commit-inst? MA))
                  (not (b1p (INST-fetch-error?
                             (inst-of-tag (MT-ROB-head MT) MT))))
```

```
                (not (b1p (INST-modified?
                           (inst-of-tag (MT-ROB-head MT) MT))))
                (not (b1p (inst-specultv?
                           (inst-of-tag (MT-ROB-head MT) MT)))))
           (equal (step-SRF MA sigs)
                  (ISA-SRF
                   (INST-post-ISA (inst-of-tag (MT-ROB-head MT) MT)))))
  :hints (("goal" :in-theory
                (enable step-SRF lift-b-ops
                        INST-exintr-INST-in-if-not-retired
                        not-INST-excpt-INST-at-ROB-head-if-not-enter-excpt)
                :use
                (:instance ISA-SRF-ISA-step-if-not-ROB-write-SRF-help)))))


; A help lemma for step-SRF-INST-post-ISA-of-INST-at-MT-ROB-head.
(defthm step-SRF-INST-post-ISA-of-INST-at-MT-ROB-head-normal-case
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (commit-inst? MA))
                (not (b1p (INST-modified?
                           (inst-of-tag (MT-ROB-head MT) MT))))
                (not (b1p (inst-specultv?
                           (inst-of-tag (MT-ROB-head MT) MT)))))
           (equal (step-SRF MA sigs)
                  (ISA-SRF
                   (INST-post-ISA (inst-of-tag (MT-ROB-head MT) MT)))))
  :hints (("goal" :cases ((b1p (INST-fetch-error?
                                (inst-of-tag (MT-ROB-head MT) MT)))))))
)

; This is a landmark lemma.
; Suppose instruction i is at the head of the ROB.  If i commits in this
; cycle, the new state of the special register file is identical to the
; special register file in the post-ISA state of i.
(defthm step-SRF-INST-post-ISA-of-INST-at-MT-ROB-head
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (commit-inst? MA))
                (not (b1p (ex-intr? MA sigs)))
                (not (MT-CMI-p (MT-step MT MA sigs))))
           (equal (step-SRF MA sigs)
                  (ISA-SRF
                   (INST-post-ISA (inst-of-tag (MT-ROB-head MT) MT)))))
  :hints (("goal"
           :cases ((b1p (INST-modified? (inst-of-tag (MT-ROB-head MT) MT)))
                   (b1p (inst-specultv? (inst-of-tag (MT-ROB-head MT) MT)))))))

(local
(defthm SRF-match-p-preserved-help
    (implies (and (MAETT-p MT) (MA-state-p MA)
                (MA-input-p sigs)
                (inv MT MA)
                (not (MT-CMI-p (MT-step MT ma sigs))))
           (equal (MT-SRF  (MT-step MT MA sigs))
                  (step-SRF MA sigs)))
  :hints (("goal" :cases ((b1p (commit-inst? MA))
                          (b1p (ex-intr? MA sigs)))))))

; SRF-match-p is preserved.
(defthm SRF-match-p-preserved
    (implies (and (MAETT-p MT) (MA-state-p MA)
```

```
                (MA-input-p sigs)
                (inv MT MA)
                (not (MT-CMI-p (MT-step MT ma sigs))))
           (SRF-match-p (MT-step MT ma sigs) (MA-step ma sigs)))
  :hints (("goal" :in-theory (enable SRF-match-p
                                     MT-CMI-p))))

(deftheory incompatible-with-excpt
    (union-theories
     '(NOT-INST-RFEH-INST-AT-ROB-HEAD-IF-NOT-LEAVE-EXCPT
      not-INST-excpt-INST-at-ROB-head-if-not-enter-excpt
      INST-EXCPT-INST-AT-ROB-HEAD-IF-ENTER-EXCPT)
     (theory 'incompatible-with-excpt-in-MAETT-lemmas)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Proof of PC-match-p preserved
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;; Proof of pc-match-p for initial states.
(defthm pc-match-p-init-MT
    (implies (MA-state-p MA)
             (pc-match-p (init-MT MA) MA))
  :Hints (("goal" :in-theory (enable pc-match-p init-mt MT-pc))))

;;;;;;; Induction case
;;; Proof by case analysis,
;;
;  Cases are
;    Case when ex-intr? is 1.
;    Case when enter-excpt? is 1
;    Case when commit-jmp? is on
;    Case when leave-excpt? is on
;    Case when fetch-inst? is on

;;;;;; Proof of pc-match-p for the case enter-excpt? is on
(defthm ISA-pc-ISA-step-ex-intr
    (implies (b1p (ISA-input-exint intr))
             (equal (ISA-pc (ISA-step ISA intr)) #x30))
  :hints (("goal" :in-theory (enable ISA-step ISA-external-intr))))


(encapsulate nil
(local
(defthm MT-pc-rob-jmp-addr-if-ex-intr-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (consp trace)
                  (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (ex-intr? MA sigs)))
             (equal (trace-pc (step-trace trace MT MA sigs
                                          (INST-pre-ISA (car trace))
                                          spc smc)
                              (ISA-pc (INST-pre-ISA (car trace))))
                    #x30))
  :hints ((when-found (INST-PRE-ISA (CAR (CDR TRACE)))
                      (:cases ((consp (cdr trace))))))
  :rule-classes nil))


; A landmark lemma.
; Next ideal pc value is supplied from ROB-jmp-addr, when an external
```

```
; interrupt occurs.
(defthm MT-pc-rob-jmp-addr-if-ex-intr
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (ex-intr? MA sigs)))
             (equal (MT-pc (MT-step MT MA sigs))
                    (ROB-jmp-addr (MA-rob MA) MA sigs)))
  :hints (("goal" :in-theory (enable ROB-jmp-addr MT-step MT-pc)
                  :use (:instance MT-pc-rob-jmp-addr-if-ex-intr-help
                                  (trace (MT-trace MT)) (spc 0) (smc 0)))))
)

;;;;;    Case when enter-excpt? is 1
(local
(encapsulate nil
(local
(defthm MT-pc-MT-step-if-INST-cause-jmp-induct
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (consp trace)
                  (member-equal i trace) (INST-p i)
                  (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (trace-all-commit-before i trace)
                  (b1p (INST-cause-jmp? i MT MA sigs)))
             (equal (trace-pc (step-trace trace MT MA sigs
                                          (INST-pre-ISA (car trace))
                                          spc smc)
                              (ISA-pc (INST-pre-ISA (car trace))))
                    (ISA-pc (INST-post-ISA i))))
  :hints ((when-found (INST-pre-ISA (car (cdr trace)))
                      (:cases ((consp (cdr trace))))))
  :rule-classes nil))

(local
(defthm MT-pc-MT-step-if-INST-cause-jmp-help
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (MT-all-commit-before i MT)
                  (b1p (INST-cause-jmp? i MT MA sigs)))
             (equal (MT-pc (MT-step MT MA sigs))
                    (ISA-pc (INST-post-ISA i))))
  :hints (("goal" :in-theory (enable MT-pc MT-step
                                     MT-all-commit-before INST-in)
          :use (:instance MT-pc-MT-step-if-INST-cause-jmp-induct
                          (trace (MT-trace MT)) (smc 0) (spc 0))))
  :rule-classes nil))

; If INST-cause-jmp? is 1, the ideal pc value for the next state is
; the PC in the post-ISA state of the instruction at the head of the ROB.
(defthm MT-pc-MT-step-if-INST-cause-jmp
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-of-tag (MT-ROB-head MT) MT)
                  (b1p (INST-cause-jmp? (inst-of-tag (MT-ROB-head MT) MT)
                                        MT MA sigs)))
     (equal (MT-pc (MT-step MT MA sigs))
            (ISA-pc (INST-post-ISA (inst-of-tag (MT-ROB-head MT) MT)))))
  :hints (("goal" :use (:instance MT-pc-MT-step-if-INST-cause-jmp-help
                                  (i (inst-of-tag (MT-ROB-head MT) MT))))))
))
```

```
(encapsulate nil
(local
(defthm ISA-pc-INST-post-ISA-INST-fetch-error
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (complete-stg-p (INST-stg i))
                  (b1p (INST-fetch-error? i))
                  (not (b1p (ISA-input-exint intr))))
             (equal (ISA-pc (ISA-step (INST-pre-ISA i) intr))
                    (logapp 4 0 (excpt-type (INST-excpt-flags i)))))
  :hints (("goal" :in-theory (enable ISA-step INST-fetch-error?
                                     ISA-fetch-error
                                     INST-excpt-flags)))
  :rule-classes nil))

(local
(defthm ISA-pc-INST-post-ISA-INST-decode-error
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (complete-stg-p (INST-stg i))
                  (not (b1p (INST-fetch-error? i)))
                  (b1p (INST-decode-error? i))
                  (not (b1p (ISA-input-exint intr))))
             (equal (ISA-pc (ISA-step (INST-pre-ISA i) intr))
                    (logapp 4 0 (excpt-type (INST-excpt-flags i)))))
  :hints (("goal" :in-theory (enable ISA-step INST-excpt-flags
                                     ISA-step-INST-pre-ISA
                                     decode-illegal-inst?
                                     ISA-MFSR ISA-MTSR
                                     INST-ra INST-rc
                                     ISA-rfeh supervisor-mode?
                                     ISA-illegal-inst
                                     INST-decode-error?
                                     lift-b-ops)))
  :rule-classes nil))

(local
(defthm ISA-pc-INST-post-ISA-INST-data-accs-error
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (complete-stg-p (INST-stg i))
                  (not (b1p (INST-fetch-error? i)))
                  (not (b1p (INST-decode-error? i)))
                  (b1p (INST-data-access-error? i))
                  (not (b1p (ISA-input-exint intr))))
             (equal (ISA-pc (ISA-step (INST-pre-ISA i) intr))
                    (logapp 4 0 (excpt-type (INST-excpt-flags i)))))
  :hints (("goal" :in-theory (enable ISA-step-INST-pre-ISA
                                     INST-data-access-error?
                                     INST-excpt-flags
                                     ISA-ldi ISA-ld
                                     ISA-st ISA-sti
                                     ISA-data-accs-error
                                     INST-load-error? INST-store-error?
                                     lift-b-ops)))
  :rule-classes nil))

; The exception vector values.
```

```
; Suppose i is an exception raising instruction.  After execution i,
; the PC is set to exception type number which is shifted to the left 4-bits.
(defthm ISA-pc-INST-post-ISA-excpt-INST
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (complete-stg-p (INST-stg i))
                  (b1p (INST-excpt? i)) (not (b1p (ISA-input-exint intr))))
             (equal (ISA-pc (ISA-step (INST-pre-ISA i) intr))
                    (logapp 4 0 (excpt-type (INST-excpt-flags i)))))
  :hints (("goal" :in-theory (enable INST-excpt? lift-b-ops)
                  :use ((:instance ISA-pc-INST-post-ISA-INST-data-accs-error)
                        (:instance ISA-pc-INST-post-ISA-INST-decode-error)
                        (:instance ISA-pc-INST-post-ISA-INST-fetch-error)))))
)

; When an exception handling occurs, INST-cause-jmp? value for the
; instruction at the head of the ROB is 1.
(defthm INST-cause-jmp-INST-at-MT-ROB-head-if-enter-excpt
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (INST-modified?
                             (inst-of-tag (MT-ROB-head MT) MT))))
                  (b1p (enter-excpt? MA)))
             (equal (INST-cause-jmp? (inst-of-tag (MT-ROB-head MT) MT)
                                     MT MA sigs)
                    1))
  :hints (("goal" :in-theory (enable enter-excpt? lift-b-ops
                                     equal-b1p-converter
                                     INST-cause-jmp?))))

; A landmark lemma.
; When an exception handling occurs, the ideal pc value at the next
; cycle is output on ROB-jmp-addr.
(defthm MT-pc-rob-jmp-addr-if-enter-excpt
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (MT-self-modify? (MT-step MT MA sigs))))
                  (b1p (enter-excpt? MA)))
             (equal (MT-pc (MT-step MT MA sigs))
                    (ROB-jmp-addr (MA-rob MA) MA sigs)))
  :hints (("goal" :cases ((b1p (INST-modified?
                                (inst-of-tag (MT-ROB-head MT) MT))))
                  :in-theory (enable ROB-jmp-addr
                                     INST-exintr-INST-in-if-not-retired))))

;;;;;; Proof of pc-match-p for the case where commit-jmp? is on

; When commit-jmp? is 1, INST-cause-jmp? is 1 for the instruction
; at the head of the ROB.
(defthm INST-cause-jmp-INST-at-MT-ROB-head-if-commit-jmp
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (INST-modified?
                             (inst-of-tag (MT-ROB-head MT) MT))))
                  (b1p (commit-jmp? MA)))
             (equal (INST-cause-jmp? (inst-of-tag (MT-ROB-head MT) MT)
                                     MT MA sigs)
                    1))
  :hints (("goal" :in-theory (e/d (enter-excpt? lift-b-ops
                                   commit-jmp? complete-stg-if-robe-complete
                                   equal-b1p-converter
```

```
                                INST-cause-jmp?)
                                  (incompatible-with-excpt)))))

; SYNC instruction increments the PC.
(defthm ISA-pc-INST-pre-ISA-if-INST-sync
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (complete-stg-p (INST-stg i))
                  (not (b1p (INST-excpt? i)))
                  (not (b1p (INST-rfeh? i)))
                  (b1p (INST-sync? i))
                  (not (b1p (ISA-input-exint intr))))
             (equal (ISA-pc (ISA-step (INST-pre-ISA i) intr))
                    (addr (1+ (ISA-pc (INST-pre-ISA i))))))
  :hints (("goal" :in-theory (enable INST-sync? ISA-step-INST-pre-ISA
                                     lift-b-ops INST-cntlv decode
                                     rdb logbit* INST-opcode
                                     INST-rfeh?
                                     ISA-sync ISA-rfeh
                                     INST-excpt?))))

; The effect of branch instruction.
(defthm ISA-pc-INST-pre-ISA-if-INST-branch
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (complete-stg-p (INST-stg i))
                  (not (b1p (INST-excpt? i)))
                  (b1p (INST-bu? i))
                  (not (b1p (ISA-input-exint intr))))
             (equal (ISA-pc (ISA-step (INST-pre-ISA i) intr))
                    (if (b1p (INST-branch-taken? i))
                        (INST-br-target i)
                        (addr (1+ (ISA-pc (INST-pre-ISA i)))))))
  :hints (("goal" :in-theory (enable ISA-step-INST-pre-ISA
                                     lift-b-ops INST-bu?
                                     INST-cntlv INST-opcode
                                     decode logbit* rdb
                                     INST-im INST-BRANCH-TAKEN?
                                     ISA-br bv-eqv-iff-equal
                                     rname-p
                                     INST-br-target
                                     INST-excpt?))))

(encapsulate nil
(local
(defthm MT-pc-rob-jmp-addr-if-commit-jmp-help-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (INST-modified?
                             (inst-of-tag (MT-rob-head MT) MT))))
                  (not (b1p (ex-intr? MA sigs)))
                  (not (b1p (leave-excpt? MA)))
                  (not (b1p (enter-excpt? MA)))
                  (b1p (commit-jmp? MA)))
             (equal (ROB-jmp-addr (MA-rob MA) MA sigs)
                    (ISA-pc (ISA-step (INST-pre-ISA
                                       (inst-of-tag (MT-ROB-head MT) MT))
                                      (ISA-input 0)))))
  :hints (("goal" :in-theory (e/d (ROB-jmp-addr
                                   enter-excpt?
```

```
                                    execute-stg-if-not-robe-complete
                                    complete-stg-if-robe-complete
                                    LEAVE-EXCPT?
                                    INST-excpt?
                                    lift-b-ops commit-jmp?)
                                  (incompatible-with-excpt))
                  :cases
                  ((b1p (INST-excpt? (inst-of-tag (MT-rob-head MT) MT))))))))

(local
(defthm MT-pc-rob-jmp-addr-if-commit-jmp-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (MT-self-modify? (MT-step MT ma sigs))))
                  (not (b1p (ex-intr? MA sigs)))
                  (not (b1p (leave-excpt? MA)))
                  (not (b1p (enter-excpt? MA)))
                  (b1p (commit-jmp? MA)))
             (equal (MT-pc (MT-step MT MA sigs))
                    (ROB-jmp-addr (MA-rob MA) MA sigs)))
  :hints (("goal" :cases ((b1p (INST-modified?
                                (inst-of-tag (MT-ROB-head MT) MT))))
                  :in-theory (enable lift-b-ops
                                     INST-exintr-INST-in-if-not-retired
                                     commit-inst?)))))

; A landmark lemma
; When commit-jmp? is on, the ideal pc value is provided through
; ROB-jmp-addr.  Commit-jmp? is on when sync or branch is executed.
(defthm MT-pc-rob-jmp-addr-if-commit-jmp
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (MT-self-modify? (MT-step MT ma sigs))))
                  (not (b1p (leave-excpt? MA)))
                  (b1p (commit-jmp? MA)))
             (equal (MT-pc (MT-step MT MA sigs))
                    (ROB-jmp-addr (MA-rob MA) MA sigs)))
  :hints (("goal" :cases ((b1p (ex-intr? MA sigs))
                          (b1p (enter-excpt? MA))))))

)

;;;;;; Proof of pc-match-p for the case leave-excpt? is on
; The effect of RFEH instruction on the PC.
(defthm ISA-pc-INST-step-if-INST-rfeh
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (INST-excpt? i)))
                  (b1p (INST-rfeh? i))
                  (not (b1p (ISA-input-exint intr))))
             (equal (ISA-pc (ISA-step (INST-pre-ISA i) intr))
                    (addr (SRF-sr0 (ISA-SRF (INST-pre-ISA i))))))
  :hints (("goal" :in-theory (enable ISA-step-INST-pre-ISA
                                     INST-rfeh? INST-cntlv INST-opcode
                                     logbit* rdb decode INST-decode-error?
                                     decode-illegal-inst? supervisor-mode?
                                     read-sreg
                                     ISA-rfeh INST-excpt?
                                     lift-b-ops))))

(encapsulate nil
```

```
(local
(defthm MT-pc-if-leave-excpt-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (INST-modified?
                             (inst-of-tag (MT-rob-head MT) MT))))
                  (not (b1p (enter-excpt? MA)))
                  (b1p (leave-excpt? MA)))
             (equal (MT-pc (MT-step MT MA sigs))
                    (addr (SRF-SR0 (MA-SRF MA))))))
  :hints (("goal" :in-theory (e/d (leave-excpt? enter-excpt?
                                   INST-excpt?
                                   INST-exintr-INST-in-if-not-retired
                                   INST-EXCPT-DETECTED-P
                                   complete-stg-if-robe-complete
                                   execute-stg-if-not-robe-complete
                                   lift-b-ops)
                                     (incompatible-with-excpt))
                  :cases
                  ((b1p (INST-excpt? (inst-of-tag (MT-rob-head MT) MT)))))))

; A landmark lemma.
; The ideal program counter value for the next state is provided from
; special register 0.
(defthm MT-pc-if-leave-excpt
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (MT-self-modify? (MT-step MT ma sigs))))
                  (not (b1p (enter-excpt? MA)))
                  (b1p (leave-excpt? MA)))
             (equal (MT-pc (MT-step MT MA sigs))
                    (addr (SRF-SR0 (MA-SRF MA))))))
  :hints (("goal" :cases ((b1p (INST-modified?
                                (inst-of-tag (MT-ROB-head MT) MT))))
                  :in-theory (enable lift-b-ops commit-inst?))))
)

;;;;;; Proof of pc-match-p for the case where no instruction is fetched.

(encapsulate nil
(local
(defthm MT-pc-if-not-fetch-inst-local
    (implies (and (inv MT MA)
                  (consp trace)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (ex-intr? MA sigs)))
                  (not (b1p (commit-jmp? MA)))
                  (not (b1p (enter-excpt? MA)))
                  (not (b1p (leave-excpt? MA)))
                  (not (b1p (fetch-inst? MA sigs))))
             (equal (trace-pc (step-trace trace MT MA sigs
                                          (INST-pre-ISA (car trace))
                                          spc smc)
                              (ISA-pc (INST-pre-ISA (car trace))))
                    (trace-pc trace (ISA-pc (INST-pre-ISA (car trace)))))))
  :hints ((when-found (INST-pre-ISA (car (cdr trace)))
                      (:cases ((consp (cdr trace)))))
          ("goal" :in-theory (enable INST-cause-jmp? lift-b-ops)))
  :rule-classes nil))
```

```
; The ideal pc value for the next cycle is the current PC value
; if no instruction is fetched.
(defthm MT-pc-if-not-fetch-inst
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (ex-intr? MA sigs)))
                  (not (b1p (commit-jmp? MA)))
                  (not (b1p (enter-excpt? MA)))
                  (not (b1p (leave-excpt? MA)))
                  (not (b1p (ex-intr? MA sigs)))
                  (not (b1p (fetch-inst? MA sigs))))
             (equal (MT-pc (MT-step MT MA sigs))
                    (MT-pc MT)))
  :hints (("goal" :use (:instance MT-pc-if-not-fetch-inst-local
                                  (trace (MT-trace MT)) (spc 0) (smc 0))
                  :in-theory (enable MT-pc MT-step))))
)

;;;;;; Proof of pc-match-p for the case fetch-inst? is on

; If the instruction at IFU is a branch instruction whose branch is
; taken, the value output on the IFU-branch-target is the ; PC in the
; post-ISA state of the branch instruction.
(defthm IFU-branch-target-INST-pc-INST-post-ISA
    (implies (and (inv MT MA)
                  (uniq-INST-at-stg '(IFU) MT)
                  (not (b1p (INST-fetch-error? (INST-at-stg '(IFU) MT))))
                  (b1p (INST-bu? (INST-at-stg '(IFU) MT)))
                  (b1p (INST-branch-taken? (INST-at-stg '(IFU) MT)))
                  (not (b1p (inst-specultv? (INST-at-stg '(IFU) MT))))
                  (not (b1p (INST-modified? (INST-at-stg '(IFU) MT))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (IFU-branch-target (MA-IFU MA))
                    (ISA-pc (INST-post-ISA (INST-at-stg '(IFU) MT)))))
  :hints (("goal" :in-theory (enable IFU-branch-target
                                     exception-relations
                                     INST-bu? INST-cntlv decode
                                     logbit* rdb INST-opcode
                                     INST-exintr-INST-in-if-not-retired
                                     INST-branch-taken? lift-b-ops
                                     ISA-br rname-p
                                     ISA-step-INST-pre-ISA))))

(encapsulate nil
(local
(defthm MT-pc-post-ISA-INST-at-IFU-induct
    (implies (and (inv MT MA)
                  (consp trace)
                  (subtrace-p trace MT)
                  (member-equal i trace) (INST-p i)
                  (IFU-stg-p (INST-stg i))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (trace-pc trace (ISA-pc (INST-pre-ISA (car trace))))
                    (ISA-pc (INST-post-ISA i))))
  :hints (("goal" :in-theory (enable INST-exintr-INST-in-if-not-retired)))
  :rule-classes nil))

(local
(defthm MT-pc-post-ISA-INST-at-IFU-help
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (IFU-stg-p (INST-stg i))
```

```
                (MAETT-p MT) (MA-state-p MA))
           (equal (MT-pc MT) (ISA-pc (INST-post-ISA i))))
  :hints (("goal" :in-theory (enable MT-pc INST-in)
                  :use (:instance MT-pc-post-ISA-INST-at-IFU-induct
                                  (trace (MT-trace MT)))))))

; When an instruction i is at IFU, the PC value calculated by MT-pc
; is the PC of the post-ISA of i.
(defthm MT-pc-post-ISA-INST-at-IFU
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(IFU) MT)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (MT-pc MT)
                    (ISA-pc (INST-post-ISA (INST-at-stg '(IFU) MT)))))
  :hints (("goal" :restrict ((MT-pc-post-ISA-INST-at-IFU-help
                              ((i (INST-at-stg '(IFU) MT))))))))
)

(in-theory (disable MT-pc-post-ISA-INST-at-IFU))

; Following lemmas are for the proof of MT-pc-if-fetch-inst.
; WARNING!!!  You can skip to MT-pc-if-fetch-inst.
; there are interesting lemmas, but some of them are about the definition
; of the speculative execution and other concepts, and the reader may be
; confused why they are related just by reading.  In fact, the
; speculative execution is closely related to the PC values, but the
; reader can get the big picture of the proof without understanding
; the following lemmas.

;
; If branch prediction and actual branch outcomes are different,
; INST-wrong-branch? of the branch instruction is set to 1.
(defthm INST-wrong-branch-step-INST-IFU
    (implies (and (inv MT MA)
                  (uniq-inst-at-stg '(IFU) MT)
                  (not (b1p (INST-branch-taken? (INST-at-stg '(IFU) MT))))
                  (b1p (IFU-branch-predict? (MA-IFU MA) MA sigs))
                  (not (b1p (DQ-full? (MA-DQ MA))))
                  (b1p (INST-bu? (INST-at-stg '(IFU) MT)))
                  (not (b1p (INST-excpt? (INST-at-stg '(IFU) MT))))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (equal (INST-wrong-branch? (step-INST (INST-at-stg '(IFU) MT)
                                                   MT MA sigs))
                    1))
  :hints (("goal" :in-theory (enable INST-wrong-branch? lift-b-ops
                                     equal-b1p-converter))))


; If a branch is wrongly predicted, the instruction starts a
; speculative execution.
(defthm INST-start-specultv-step-INST-IFU
    (implies (and (inv MT MA)
                  (not (b1p (INST-branch-taken? (INST-at-stg '(IFU) MT))))
                  (b1p (IFU-branch-predict? (MA-IFU MA) MA sigs))
                  (not (b1p (DQ-full? (MA-DQ MA))))
                  (not (b1p (inst-specultv? (INST-at-stg '(IFU) MT))))
                  (not (b1p (INST-modified? (INST-at-stg '(IFU) MT))))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (equal (INST-start-specultv? (step-INST (INST-at-stg '(IFU) MT)
                                                     MT MA sigs))
                    1))
  :hints (("goal" :in-theory (enable INST-start-specultv? lift-b-ops
```

```
                                    INST-BU-OPCODE-2
                                    INST-opcode
                                    equal-b1p-converter
                                    IFU-branch-predict?)
                  :cases ((b1p (INST-excpt? (INST-at-stg '(IFU) MT)))))))

; If the branch prediction is incorrect for the instruction at IFU,
; the processor starts speculative execution of instructions.
(defthm not-MT-specultv-if-correct-branch-predict
    (implies (and (inv MT MA)
                  (not (b1p (DQ-full? (MA-DQ MA))))
                  (not (b1p (INST-branch-taken? (INST-at-stg '(IFU) MT))))
                  (not (b1p (inst-specultv? (INST-at-stg '(IFU) MT))))
                  (not (b1p (INST-modified? (INST-at-stg '(IFU) MT))))
                  (b1p (IFU-branch-predict? (MA-IFU MA) MA sigs))
                  (not (b1p (flush-all? MA sigs)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (equal (MT-specultv? (MT-step MT MA sigs)) 1))
  :hints (("goal" :restrict
                  ((MT-inst-specultv-if-INST-start-specultv
                    ((i (step-INST (INST-at-stg '(IFU) MT) MT MA sigs)))))
                  :in-theory (enable IFU-branch-predict?
                                     lift-b-ops))))

; If a fetch error occurs, a speculative execution starts.
; Note: even after a fetch error, the processor continues to
; fetch instructions.  We consider this as speculative actions.
(defthm MT-specultv-if-INST-fetch-error
    (implies (and (b1p (INST-fetch-error? i))
                  (INST-in i MT) (INST-p i)
                  (not (dispatched-p i))
                  (not (b1p (flush-all? MA sigs))))
             (equal (MT-specultv? (MT-step MT MA sigs)) 1))
  :Hints (("goal" :restrict ((MT-INST-SPECULTV-IF-INST-START-SPECULTV
                              ((i (step-INST i MT MA sigs)))))
                  :in-theory (enable INST-start-specultv?
                                     lift-b-ops dispatched-p
                                     INST-excpt?))))

(local
(defthm IFU-branch-target-if-IFU-branch-predict
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (b1p (IFU-branch-predict? (MA-IFU MA) MA sigs))
                  (not (b1p (enter-excpt? MA)))
                  (not (b1p (ex-intr? MA sigs)))
                  (not (b1p (leave-excpt? MA)))
                  (not (b1p (commit-jmp? MA)))
                  (not (b1p (DQ-full? (MA-DQ MA))))
                  (not (b1p (MT-specultv? (MT-step MT MA sigs))))
                  (not (b1p (MT-self-modify? MT))))
             (equal (MT-pc MT)
                    (IFU-branch-target (MA-IFU MA))))
  :hints (("goal" :in-theory (enable IFU-branch-predict? lift-b-ops
                                     INST-BU-OPCODE-2 INST-opcode
                                     exception-relations
                                     MT-pc-post-ISA-INST-at-IFU
                                     flush-all?)
                  :cases ((b1p (inst-specultv? (INST-at-stg '(IFU) MT)))
                          (b1p (INST-modified? (INST-at-stg '(IFU) MT)))
                          (b1p (INST-fetch-error? (INST-at-stg '(IFU) MT))))))
```

```
          ("subgoal 4" :cases ((not (b1p (INST-branch-taken?
                                          (INST-at-stg '(IFU) MT))))))))

(encapsulate nil
(local
(defthm MT-pc-MT-step-if-fetched-inst-help
    (implies (b1p (fetch-inst? MA sigs))
             (equal (trace-pc (step-trace trace MT MA sigs
                                          ISA spc smc)
                              (ISA-pc ISA))
                    (ISA-pc (INST-post-ISA
                             (fetched-inst MT (trace-final-ISA trace ISA)
                                           spc1 smc1)))))))

; When an instruction is fetched, the ideal PC value is the PC in the
; post-ISA state of the fetched instruction.
(defthm MT-pc-MT-step-if-fetched-inst
    (implies (b1p (fetch-inst? MA sigs))
             (equal (MT-pc (MT-step MT MA sigs))
                    (ISA-pc (INST-post-ISA
                             (fetched-inst MT (MT-final-ISA MT)
                                           (MT-specultv? MT)
                                           (MT-self-modify? MT))))))
  :hints (("goal" :in-theory (enable MT-pc MT-step
                                     MT-final-ISA)
                  :use (:instance MT-pc-MT-step-if-fetched-inst-help
                                  (trace (MT-trace MT))
                                  (spc1 (MT-specultv? MT))
                                  (smc1 (MT-self-modify? MT))
                                  (ISA (MT-init-ISA MT))))))
)

; This is a technical lemma, and the reader may want to skip this.
; The ISA-start-specultv becomes 1, whenever the machine fetches an
; instruction that will not be committed.  For instance, such
; instructions are fetched after a mispredicted branch instruction, an
; exception-raising instruction, and so on.  Otherwise, the PC is
; incremented by 1.
(defthm ISA-pc-INST-pre-ISA-if-not-INST-start-specultv
    (implies (and (not (b1p (INST-start-specultv?
                             (fetched-inst MT ISA spc smc))))
                  (not (b1p (ISA-input-exint intr))))
             (equal (ISA-pc (ISA-step ISA intr))
                    (addr (+ 1 (ISA-pc ISA)))))
  :hints (("goal" :in-theory (enable lift-b-ops INST-excpt?
                                     ISA-step
                                     INST-wrong-branch?
                                     INST-context-sync? INST-sync?
                                     INST-fetch-error?
                                     INST-decode-error?
                                     decode-illegal-inst?
                                     supervisor-mode? INST-ra INST-rc
                                     INST-data-access-error?
                                     INST-load-error? INST-store-error?
                                     INST-branch-taken?
                                     INST-bu? INST-cntlv INST-opcode
                                     decode logbit* rdb
                                     ISA-add ISA-mul ISA-br
                                     ISA-st ISA-sti ISA-MFSR
                                     ISA-MTSR
                                     ISA-ld ISA-ldi
                                     INST-start-specultv?))))
```

```
(encapsulate nil
(local
(defthm member-equal-fetched-inst
   (implies (and (inv MT MA)
                 (b1p (fetch-inst? MA sigs))
                 (subtrace-p trace MT) (INST-listp trace)
                 (MAETT-p MT) (MA-state-p MA))
    (member-equal (fetched-inst MT (trace-final-ISA trace
                                                    (ISA-before trace MT))
                                (MT-specultv? MT)
                                (MT-self-modify? MT))
                  (step-trace trace MT MA sigs
                              (ISA-before trace MT)
                              (specultv-before? trace MT)
                              (modified-inst-before? trace MT))))
  :hints (("goal" :in-theory (enable lift-b-ops)))
  :rule-classes nil))

; The fetched instruction is in the new MAETT.
(defthm INST-in-fetched-inst
    (implies (and (inv MT MA)
                  (b1p (fetch-inst? MA sigs))
                  (MAETT-p MT) (MA-state-p MA))
    (INST-in (fetched-inst MT (MT-final-ISA MT) (MT-specultv? MT)
                           (MT-self-modify? MT))
             (MT-step MT MA sigs)))
  :hints (("goal" :use (:instance member-equal-fetched-inst
                                  (trace (MT-trace MT)))
                  :in-theory (enable MT-final-ISA MT-specultv? INST-in
                                     MT-self-modify? MT-step))))
)

; If the fetched instruction starts speculative execution,
; MT-specultv? is 1. (That means the processor is executing speculatively.)
(defthm INST-in-specultv-if-INST-start-specultv-fetched-inst
    (implies (and (inv MT MA)
                  (b1p (fetch-inst? MA sigs))
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (INST-start-specultv?
                        (fetched-inst MT (MT-final-ISA MT)
                                      (MT-specultv? MT)
                                      (MT-self-modify? MT)))))
             (equal (MT-specultv? (MT-step MT MA sigs)) 1))
  :hints (("goal" :in-theory (enable fetch-inst? lift-b-ops)
                  :restrict
                  ((MT-INST-SPECULTV-IF-INST-START-SPECULTV
                    ((i (fetched-inst MT (MT-final-ISA MT)
                                      (MT-specultv? MT)
                                      (MT-self-modify? MT)))))))))

(encapsulate nil
(local
(defthm MT-specultv-MT-step-if-fetch-inst-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (b1p (fetch-inst? MA sigs))
                  (not (b1p (trace-self-modify? trace)))
                  (b1p (trace-specultv? trace))
                  (INST-listp trace) (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs))
             (equal (trace-specultv? (step-trace trace MT MA sigs
```

```
                                                ISA spc smc))
                   1))
  :hints (("goal" :in-theory (enable lift-b-ops fetch-inst?)))))

; If the processor fetches an instruction, and it has been executing
; instructions speculatively, it continues to execute instructions
; speculatively.
(defthm MT-specultv-MT-step-if-fetch-inst
    (implies (and (inv MT MA)
                  (b1p (fetch-inst? MA sigs))
                  (not (b1p (MT-self-modify? MT)))
                  (b1p (MT-specultv? MT))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (equal (MT-specultv? (MT-step MT MA sigs)) 1))
  :hints (("goal" :in-theory (enable MT-step MT-specultv? MT-self-modify?))))

)

(encapsulate nil
(local
(defthm MT-self-modify-MT-step-if-fetch-inst-help
    (implies (and (INST-listp trace)
                  (b1p (trace-self-modify? trace))
                  (b1p (fetch-inst? MA sigs)))
             (equal (trace-self-modify? (step-trace trace MT MA sigs
                                                    ISA spc smc)) 1))
  :hints (("goal" :in-theory (enable fetch-inst? lift-b-ops)))))

; The processor continues to execute a modified stream of instructions,
; if it fetches an instruction this cycle.
(defthm MT-self-modify-MT-step-if-fetch-inst
    (implies (and (MAETT-p MT) (b1p (MT-self-modify? MT))
                  (b1p (fetch-inst? MA sigs)))
             (equal (MT-self-modify? (MT-step MT MA sigs)) 1))
  :hints (("goal" :in-theory (enable MT-self-modify? MT-step))))
)

; A landmark lemma.  When the processor fetches an instruction,
; the pc is incremented.
(defthm MT-pc-if-fetch-inst
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (not (b1p (MT-specultv? (MT-step MT MA sigs))))
                  (not (b1p (MT-self-modify? (MT-step MT MA sigs))))
                  (b1p (fetch-inst? MA sigs)))
             (equal (MT-pc (MT-step MT MA sigs))
                    (addr (+ 1 (MA-pc MA)))))
  :hints (("goal" :in-theory
                  (e/d ()
                       (ISA-pc-INST-pre-ISA-if-not-INST-start-specultv)
                       )
                  :cases ((b1p (MT-self-modify? MT))
                          (b1p (INST-start-specultv?
                                 (FETCHED-INST MT (MT-FINAL-ISA MT)
                                               (MT-SPECULTV? MT)
                                               (MT-SELF-MODIFY? MT))))))
          ("subgoal 3" :cases ((b1p (MT-specultv? MT))))
          ("subgoal 3.2" :use (:instance
                               ISA-pc-INST-pre-ISA-if-not-INST-start-specultv
                               (intr (ISA-input 0))
                               (ISA (MT-final-ISA MT))
```

```
                            (spc (MT-specultv? MT))
                            (smc (MT-self-modify? MT))))))

;; Now all cases are covered, following are the lemmas to help the proof of
;; pc-match-p-preserved.
(encapsulate nil
(local
(defthm MT-self-modify-MT-step-help
    (implies (and (INST-listp trace)
                  (not (b1p (ex-intr? MA sigs)))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (fetch-inst? MA sigs))))
             (equal (trace-self-modify? (step-trace trace MT MA sigs
                                                    ISA spc smc))
                    (trace-self-modify? trace)))
  :hints (("goal" :in-theory (enable flush-all? lift-b-ops ex-intr?
                                     INST-cause-jmp? INST-exintr-now?)))))

; If a processor is executing a modified stream of instructions,
; it will continue to execute modified instruction stream unless
; some the pipeline is flushed.
(defthm MT-self-modify-MT-step
    (implies (and (MAETT-p MT)
                  (not (b1p (ex-intr? MA sigs)))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (fetch-inst? MA sigs))))
             (equal (MT-self-modify? (MT-step MT MA sigs))
                    (MT-self-modify? MT)))
  :hints (("goal" :in-theory (enable MT-self-modify? MT-step))))
)

; Following are several lemmas about speculative execution,
; branching and exceptions.
(encapsulate nil
(local
(defthm MT-specultv-MT-step-if-not-branch-predict-induct
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (fetch-inst? MA sigs)))
                  (not (b1p (trace-self-modify? trace)))
                  (not (b1p (IFU-branch-predict? (MA-IFU MA) MA sigs))))
             (equal (trace-specultv? (step-trace trace MT MA sigs
                                                 ISA spc smc))
                    (trace-specultv? trace)))
  :hints (("goal" :in-theory (enable lift-b-ops)))))

(defthm MT-specultv-MT-step-if-not-branch-predict
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (MT-self-modify? MT)))
                  (not (b1p (fetch-inst? MA sigs)))
                  (not (b1p (IFU-branch-predict? (MA-IFU MA) MA sigs))))
             (equal (MT-specultv? (MT-step MT MA sigs))
                    (MT-specultv? MT)))
  :hints (("goal" :in-theory (enable MT-specultv? MT-step
                                     MT-self-modify?))))
)
```

```
(local
(encapsulate nil
(local
(defthm MT-specultv-MT-step-if-dq-full-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (fetch-inst? MA sigs)))
                  (not (b1p (trace-self-modify? trace)))
                  (b1p (DQ-full? (MA-DQ MA))))
             (equal (trace-specultv? (step-trace trace MT MA sigs
                                                 ISA spc smc))
                    (trace-specultv? trace)))
  :hints (("goal" :in-theory (enable lift-b-ops)))))

(defthm MT-specultv-MT-step-if-dq-full
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (fetch-inst? MA sigs)))
                  (not (b1p (MT-self-modify? MT)))
                  (b1p (DQ-full? (MA-DQ MA))))
             (equal (MT-specultv? (MT-step MT MA sigs))
                    (MT-specultv? MT)))
  :hints (("goal" :in-theory (enable MT-specultv? MT-step
                                     MT-self-modify?))))
))

(local
(defthm INST-fetch-error-fetched-inst-if-IFU-fetch-prohibited
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (MT-specultv? MT)))
                  (not (b1p (MT-self-modify? MT)))
                  (b1p (IFU-fetch-prohibited? (MA-pc MA) (MA-mem MA)
                                              (SRF-su (MA-SRF MA)))))
             (equal (INST-fetch-error? (fetched-inst MT
                                                     (MT-final-ISA MT)
                                                     spc smc))
                    1))
  :hints (("goal" :in-theory (enable INST-fetch-error? lift-b-ops
                                     IFU-fetch-prohibited?
                                     read-error?
                                     equal-b1p-converter)))))

(local
(defthm INST-excpt-fetched-inst-if-IFU-fetch-prohibited
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (MT-specultv? MT)))
                  (not (b1p (MT-self-modify? MT)))
                  (b1p (IFU-fetch-prohibited? (MA-pc MA) (MA-mem MA)
                                              (SRF-su (MA-SRF MA)))))
             (equal (INST-excpt? (fetched-inst MT
                                               (MT-final-ISA MT)
                                               spc smc))
                    1))
  :hints (("goal" :in-theory (enable INST-excpt?)))))
```

```
(local
(defthm INST-start-specultv-fetched-inst-if-IFU-fetch-prohibited
    (implies (and (inv MT MA)
                  (not (b1p (MT-specultv? MT)))
                  (not (b1p (MT-self-modify? MT)))
                  (b1p (IFU-fetch-prohibited? (MA-pc MA) (MA-mem MA)
                                              (SRF-su (MA-SRF MA))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-start-specultv? (fetched-inst MT
                                                        (MT-final-ISA MT)
                                                        spc smc)) 1))
  :hints (("goal" :in-theory (enable INST-start-specultv?)))))

(local
(encapsulate nil
(local
(defthm MT-specultv-MT-step-if-fetch-prohibited-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (not (b1p (specultv-before? trace MT)))
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (IFU-fetch-prohibited? (MA-pc MA) (MA-mem MA)
                                              (SRF-su (MA-SRF MA))))
                  (b1p (fetch-inst? MA sigs))
                  (not (b1p (MT-self-modify? MT))))
             (equal (trace-specultv? (step-trace trace MT MA sigs
                                                 (ISA-before trace MT)
                                                 spc smc))
                    1))
  :hints (("goal" :in-theory (enable lift-b-ops)))
  :rule-classes nil))

(defthm MT-specultv-MT-step-if-fetch-prohibited
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (IFU-fetch-prohibited? (MA-pc MA) (MA-mem MA)
                                              (SRF-su (MA-SRF MA))))
                  (b1p (fetch-inst? MA sigs))
                  (not (b1p (MT-self-modify? MT))))
             (equal (MT-specultv? (MT-step MT MA sigs)) 1))
  :hints (("goal" :in-theory (enable MT-specultv? MT-step)
                  :use (:instance
                        MT-specultv-MT-step-if-fetch-prohibited-help
                        (trace (MT-trace MT))
                        (spc 0) (smc 0)))))
))


(local
(defthm pc-match-p-preserved-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (not (b1p (MT-specultv? (MT-step MT MA sigs))))
                  (not (b1p (MT-self-modify? (MT-step MT MA sigs)))))
             (equal (MT-pc (MT-step MT MA sigs)) (step-pc MA sigs)))
  :hints (("goal" :in-theory (e/d (step-pc lift-b-ops fetch-inst?
                                   MT-specultv-p MT-self-modify-p
                                     IFU-branch-predict?
                                     flush-all?)
```

```
                                    (MT-PC-MT-STEP-IF-FETCHED-INST
                                     INST-POST-ISA-FETCHED-INST
                                     MT-PC-MT-STEP-IF-INST-CAUSE-JMP
                                     INST-CAUSE-JMP-IF-LEAVE-EXCPT))
                  :cases ((b1p (fetch-inst? MA sigs))))
          ("subgoal 1" :cases ((b1p (MT-self-modify? MT)))))))

; pc-match-p is an invariant.
(defthm pc-match-p-preserved
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (pc-match-p (MT-step MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable pc-match-p lift-b-ops
                                     MT-specultv-p MT-self-modify-p))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Proof of Mem-match-p
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Proof of mem-match-p for initial state
(defthm mem-match-p-init-MT
    (implies (MA-state-p MA)
             (mem-match-p (init-MT MA) MA))
  :hints (("goal" :in-theory (enable mem-match-p init-MT MT-mem))))

;;; Proof of induction case
;; There are two landmark lemmas.
;;    MT-mem-if-step-MT-if-release-wbuf0
;;    MT-mem-if-step-MT-if-not-release-wbuf0
; These lemmas define the value of MT-mem in the next cycle.
; The basic idea of proof is comparing
;  (step-mem MA sigs) and (MT-mem (MT-step MT MA sigs)).
; Both sides are expressed in terms of MA, and proven to be equivalent.

; An external interrupt does not change the memory.
(defthm ISA-mem-ISA-step-if-exint
    (implies (b1p (ISA-input-exint intr))
             (equal (ISA-mem (ISA-step ISA intr)) (ISA-mem ISA)))
  :hints (("goal" :in-theory (enable ISA-step ISA-external-intr))))

; An instruction causing an internal exception does not modify the memory.
(defthm ISA-mem-ISA-step-if-INST-excpt
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (b1p (INST-excpt? i))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (ISA-mem (ISA-step (INST-pre-ISA i) intr))
                    (ISA-mem (INST-pre-ISA i))))
  :hints (("goal" :in-theory (enable INST-excpt? lift-b-ops
                                     ISA-external-intr
                                     INST-fetch-error? INST-decode-error?
                                     INST-data-access-error?
                                     INST-store-error? INST-load-error?
                                     decode-illegal-inst?
                                     ISA-fetch-error
                                     ISA-step-functions
                                     ISA-step))))

; If i is not a store instruction, i does not modifier the memory.
(defthm ISA-mem-ISA-step-if-not-INST-store
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
```

```
                (not (b1p (INST-store? i))))
            (equal (ISA-mem (ISA-step (INST-pre-ISA i) intr))
                   (ISA-mem (INST-pre-ISA i))))
  :hints (("goal" :in-theory (enable ISA-step INST-store? lift-b-ops
                                     INST-ld-st? INST-cntlv
                                     INST-LSU?
                                     INST-opcode decode logbit* rdb
                                     ISA-external-intr
                                     ISA-fetch-error
                                     ISA-add ISA-mul ISA-br
                                     ISA-data-accs-error
                                     ISA-ld ISA-ldi
                                     ISA-illegal-INST ISA-mfsr ISA-mtsr
                                     ISA-sync ISA-rfeh))))

; If an exception occurs this cycle, then the instruction at the
; head of the ROB does not modify the memory.
(defthm ISA-mem-INST-step-if-enter-excpt
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (equal (inst-tag i) (MT-ROB-head MT))
                  (complete-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (b1p (enter-excpt? MA)))
             (equal (ISA-mem (ISA-step (INST-pre-ISA i) ISA-sigs))
                    (ISA-mem (INST-pre-ISA i))))
  :hints (("goal" :in-theory (enable enter-excpt? lift-b-ops
                                     exception-relations)
                  :cases ((b1p (INST-excpt? i))))))

; If commit-jmp? is on, the instruction at the head of the ROB does not
; modify the memory.
; See ISA-mem-INST-step-if-INST-cause-jmp.
(defthm ISA-mem-INST-step-if-commit-jmp
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (equal (inst-tag i) (MT-ROB-head MT))
                  (complete-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (b1p (commit-jmp? MA)))
             (equal (ISA-mem (ISA-step (INST-pre-ISA i) ISA-sigs))
                    (ISA-mem (INST-pre-ISA i))))
  :hints (("goal" :cases ((not (b1p (INST-store? i))) (b1p (INST-excpt? i)))
                  :in-theory (enable commit-jmp? lift-b-ops
                                     INST-excpt?))))

; If leave-excpt? is on, then the instruction at the head of the ROB does
; not modify the memory.
; See ISA-mem-INST-step-if-INST-cause-jmp.
(defthm ISA-mem-INST-step-if-leave-excpt
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (equal (inst-tag i) (MT-ROB-head MT))
                  (complete-stg-p (INST-stg i))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i)))
                  (b1p (leave-excpt? MA)))
```

```
            (equal (ISA-mem (ISA-step (INST-pre-ISA i) ISA-sigs))
                   (ISA-mem (INST-pre-ISA i))))
  :hints (("goal" :cases ((not (b1p (INST-store? i))) (b1p (INST-excpt? i)))
                  :in-theory (enable leave-excpt? lift-b-ops
                                     INST-excpt?))))

; INST-cause-jmp? is on for i, if i does not modify the memory.
; An instruction causing a jump (in any sense) does not modify a memory.
; For instance, a branch instruction or a RFEH instruction does not modify
; the memory.
(defthm ISA-mem-INST-step-if-INST-cause-jmp
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (not (MT-CMI-p (MT-step MT MA MA-sigs)))
                  (b1p (INST-cause-jmp? i MT MA MA-sigs)))
             (equal (ISA-mem (ISA-step (INST-pre-ISA i) ISA-sigs))
                    (ISA-mem (INST-pre-ISA i))))
  :hints (("goal" :in-theory (enable INST-cause-jmp? lift-b-ops)
                  :cases ((b1p (inst-specultv? i))
                          (b1p (INST-modified? i))))))

; If i is a store instruction, it does not cause a jump (in any sense).
(defthm not-INST-cause-jmp-if-INST-store
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (MT-CMI-p (MT-step MT MA MA-sigs)))
                  (b1p (INST-store? i)) (not (b1p (INST-excpt? i))))
             (equal (INST-cause-jmp? i MT MA sigs) 0))
  :hints (("goal" :in-theory (e/d (INST-cause-jmp? lift-b-ops
                                     INST-excpt? commit-jmp?
                                     commit-inst?
                                     enter-excpt? leave-excpt?
                                     equal-b1p-converter)
                                  (incompatible-with-excpt))
                  :cases ((b1p (INST-modified? i))
                          (b1p (inst-specultv? i))))))

(local
(defthm retire-stg-p-step-INST-if-not-INST-store
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (INST-store? i)))
                  (b1p (INST-cause-jmp? i MT MA sigs))
                  (not (MT-CMI-p (MT-step MT MA MA-sigs))))
             (retire-stg-p (INST-stg (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (enable complete-stg-p
                                     step-inst step-inst-complete
                                     INST-cause-jmp? INST-commit?
                                     lift-b-ops)
                  :cases ((b1p (inst-specultv? i))
                          (b1p (INST-modified? i)))))))

; If i causes an exception, it retires immediately.
(defthm retire-stg-p-step-INST-if-INST-excpt
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (INST-excpt? i))
                  (b1p (INST-cause-jmp? i MT MA sigs))
```

```
                (not (MT-CMI-p (MT-step MT MA MA-sigs))))
            (retire-stg-p (INST-stg (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (e/d (step-inst step-inst-complete
                                   complete-stg-p
                                   lift-b-ops INST-commit?
                                  INST-cause-jmp?))
                :cases ((b1p (inst-specultv? i))
                        (b1p (INST-modified? i))))))

; If i causes a jump, i retires immediately.
(defthm retire-stg-p-step-inst-if-INST-cause-jmp
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (INST-cause-jmp? i MT MA sigs))
                (not (MT-CMI-p (MT-step MT MA MA-sigs))))
            (retire-stg-p (INST-stg (step-INST i MT MA sigs))))
  :hints (("goal" :cases ((b1p (INST-excpt? i)) (not (b1p (INST-store? i)))))))

; If instruction i directly retires without going through the commit stage,
; i is not a store instruction and the memory is not updated.
(defthm not-retire-stg-p-step-inst-if-INST-store
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA)
                (complete-stg-p (INST-stg i))
                (retire-stg-p (INST-stg (step-INST i MT MA sigs)))
                (not (b1p (INST-modified? i)))
                (not (b1p (inst-specultv? i))))
            (equal (ISA-mem (ISA-step (INST-pre-ISA i) ISA-sigs))
                   (ISA-mem (INST-pre-ISA i))))
  :hints (("goal" :in-theory (enable complete-stg-p lift-b-ops
                                     NOT-INST-STORE-IF-COMPLETE
                                     INST-excpt? INST-COMMIT?
                                     enter-excpt?
                                     step-inst step-inst-complete)
                :cases ((b1p (INST-excpt? i)) (not (b1p (INST-store? i)))))))

; If i is in the write buffer but not at wbuf0, then i will stay in
; the commit stage.
(defthm commit-stg-p-step-INST-if-not-release-wbuf0
    (implies (and (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                (commit-stg-p (INST-stg i)))
            (commit-stg-p (INST-stg (step-INST i MT MA sigs))))
  :hints (("goal" :in-theory (enable step-INST step-INST-commit))))

(encapsulate nil
(local
(defthm ISA-mem-ISA-step-if-not-release-wbuf0-help
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (INST-in i MT) (INST-p i)
                (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                (not (retire-stg-p (INST-stg i)))
                (not (b1p (INST-modified? i)))
                (retire-stg-p (INST-stg (step-INST i MT MA sigs))))
            (equal (inst-specultv? i) 0))
  :hints (("goal"
           :use ((:instance INST-is-at-one-of-the-stages)
                 (:instance
                  inst-specultv-inst-at-rob-head-if-uniq-inst-at-rob-head))
           :in-theory
```

```
           (e/d (step-INST-complete-INST
                 INST-commit? lift-b-ops
                 step-inst-complete)
                (INST-is-at-one-of-the-stages
                 inst-specultv-inst-at-rob-head-if-uniq-inst-at-rob-head))))))

(local
(defthm ISA-mem-ISA-step-if-not-release-wbuf0-help2
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                  (not (retire-stg-p (INST-stg i)))
                  (retire-stg-p (INST-stg (step-INST i MT MA sigs)))
                  (not (MT-CMI-p (MT-step MT MA sigs))))
             (equal (INST-modified? i) 0))
  :hints (("goal" :use ((:instance INST-is-at-one-of-the-stages))
                  :in-theory (e/d (step-INST-complete-INST
                                   lift-b-ops
                                   equal-b1p-converter
                                   step-inst-complete)
                                  (inst-is-at-one-of-the-stages))))))

; If instruction i is not at wbuf0, and i advances to the retire stage,
; then the execution of i does not modify the memory, because i cannot
; be a store instruction.
(defthm ISA-mem-ISA-step-if-not-release-wbuf0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (release-wbuf0? (MA-LSU MA) MA-sigs)))
                  (not (retire-stg-p (INST-stg i)))
                  (retire-stg-p (INST-stg (step-INST i MT MA MA-sigs)))
                  (not (MT-CMI-p (MT-step MT MA MA-sigs))))
             (equal (ISA-mem (ISA-step (INST-pre-ISA i) ISA-sigs))
                    (ISA-mem (INST-pre-ISA i))))
  :hints (("goal" :use (:instance INST-is-at-one-of-the-stages)
                  :in-theory (disable INST-is-at-one-of-the-stages))))
)

(encapsulate nil
(local
(defthm MT-mem-local-help1
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (consp trace)
                  (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                  (not (retire-stg-p (INST-stg (car trace))))
                  (equal (ISA-mem ISA) mem)
                  (retire-stg-p (INST-stg (step-INST (car trace) MT MA sigs)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (trace-mem (step-trace (cdr trace) MT MA sigs ISA spc smc)
                               mem)
                    mem))
  :hints (("goal" :cases ((consp (cdr trace)))
                  :expand (STEP-TRACE (CDR TRACE)
                                      MT MA SIGS ISA SPC SMC))
          ("subgoal 1" :use
                       ((:instance
                         NOT-RETIRE-STG-P-STEP-INST-IF-EARLIER-INST-COMMIT
                         (j (car trace)) (i (cadr trace)))
```

```
                      (:instance
                       INST-is-at-one-of-the-stages (i (car trace))))
                     :in-theory
                     (e/d (step-INST-complete-INST step-INST-complete
                           COMMITTED-P lift-b-ops)
                          (NOT-RETIRE-STG-P-STEP-INST-IF-EARLIER-INST-COMMIT
                           INST-is-at-one-of-the-stages))))))

(local
(defthm MT-mem-if-step-MT-if-not-release-wbuf0-help
    (implies (and (inv MT MA)
                  (consp trace)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (not (b1p (release-wbuf0? (MA-LSU MA) sigs))))
             (equal (trace-mem (step-trace trace MT MA sigs
                                           (INST-pre-ISA (car trace))
                                           spc smc)
                               (ISA-mem (INST-pre-ISA (car trace))))
                    (trace-mem trace (ISA-mem (INST-pre-ISA (car trace))))))
  :hints (("goal" :induct t
                  :expand ((TRACE-MEM (LIST (STEP-INST (CAR TRACE) MT MA SIGS))
                                      (ISA-MEM (INST-PRE-ISA (CAR TRACE))))))
          (when-found (INST-pre-ISA (car (cdr trace)))
                      (:cases ((consp (cdr trace))))))
  :rule-classes nil))

; a landmark lemma.
; If the instruction at wbuf0 is not released, the ideal memory
; state does not change.
(defthm MT-mem-if-step-MT-if-not-release-wbuf0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (not (b1p (release-wbuf0? (MA-LSU MA) sigs))))
             (equal (MT-mem (MT-step MT MA sigs))
                    (MT-mem MT)))
  :hints (("goal" :cases ((consp (MT-trace MT)))
                  :in-theory (e/d (MT-mem MT-step)
                                  (MT-MEM-=-MA-MEM)))
          ("subgoal 1"
           :use (:instance MT-mem-if-step-MT-if-not-release-wbuf0-help
                           (trace (MT-trace MT))
                           (spc 0) (smc 0)))))
)

(local
(encapsulate nil
(local
(defthm no-inst-at-stgs-wbuf0-if-car-at-commit-wbuf0-help1
    (implies (and (member-equal (INST-stg (car sub)) stgs)
                  (tail-p sub trace)
                  (uniq-inst-at-stgs-in-trace stgs trace))
             (no-inst-at-stgs-in-trace stgs (cdr sub)))))

(local
(defthm no-inst-at-stgs-wbuf0-if-car-at-commit-wbuf0-help2
    (implies (and (tail-p sub trace)
                  (consp sub)
                  (no-inst-at-stgs-in-trace stgs trace))
```

```
            (no-inst-at-stgs-in-trace stgs (cdr sub)))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     NO-LSU-STG-CONFLICT)))))

; A help lemma.
(defthm no-inst-at-stgs-wbuf0-if-car-at-commit-wbuf0
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (consp trace)
                  (equal (INST-stg (car trace)) '(commit wbuf0))
                  (equal stgs '((LSU wbuf0)
                                (LSU wbuf0 lch)
                                (complete wbuf0)
                                (commit wbuf0))))
             (no-inst-at-stgs-in-trace stgs (cdr trace)))
  :hints (("goal" :in-theory (enable inv no-stage-conflict
                                     subtrace-p
                                     no-inst-at-stgs
                                     uniq-inst-at-stgs
                                     NO-LSU-STG-CONFLICT))))
))

(in-theory (enable inst-stg-step-inst))

(encapsulate nil
(local
 (defthm ISA-mem-ISA-step-if-non-retire-retires-help2
     (implies (and (inv MT MA)
                   (INST-in i MT) (INST-p i)
                   (not (retire-stg-p (INST-stg i)))
                   (not (equal (INST-stg i) '(commit wbuf0)))
                   (b1p (inst-specultv? i))
                   (MAETT-p MT) (MA-state-p MA))
              (not (retire-stg-p (INST-stg (step-INST i MT MA sigs)))))
   :hints (("goal" :use (:instance inst-is-at-one-of-the-stages (i i))
                   :in-theory (e/d (commit-stg-p)
                                   (inst-is-at-one-of-the-stages))))))

(local
 (defthm ISA-mem-ISA-step-if-non-retire-retires-help3
     (implies (and (inv MT MA)
                   (INST-in i MT) (INST-p i)
                   (b1p (INST-modified? i))
                   (not (retire-stg-p (INST-stg i)))
                   (not (equal (INST-stg i) '(commit wbuf0)))
                   (retire-stg-p (INST-stg (step-INST i MT MA sigs)))
                   (MAETT-p MT) (MA-state-p MA))
              (MT-CMI-p (MT-step MT MA sigs)))
   :hints (("goal" :use (:instance inst-is-at-one-of-the-stages (i i))
                   :in-theory (e/d (commit-stg-p complete-stg-p
                                      lift-b-ops)
                                   (inst-is-at-one-of-the-stages))))))


(local
(defthm ISA-mem-ISA-step-if-non-retire-retires-help
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (retire-stg-p (INST-stg i)))
                  (not (equal (INST-stg i) '(commit wbuf0)))
                  (retire-stg-p (INST-stg (step-inst i MT MA MA-sigs)))
                  (not (b1p (inst-specultv? i)))
```

```
                (not (b1p (INST-modified? i)))
                (MAETT-p MT) (MA-state-p MA))
           (equal (ISA-mem (ISA-step (INST-pre-ISA i) intr))
                  (ISA-mem (INST-pre-ISA i))))
  :hints (("goal" :cases ((b1p (INST-store? i))))
          ("subgoal 1" :cases  ((b1p (INST-excpt? i))))
          ("subgoal 1.2" :use (:instance
                               inst-is-at-one-of-the-stages (i i))
                         :in-theory (e/d (commit-stg-p lift-b-ops
                                          NOT-INST-STORE-IF-COMPLETE
                                          INST-excpt?
                                          complete-stg-p)
                                         (inst-is-at-one-of-the-stages))))))

; If any instruction goes to the retire stage from a stage other than
; (commit wbuf0), then the instruction is not a store instruction.
; Thus, the instruction has no effect on the memory.
(defthm ISA-mem-ISA-step-if-non-retire-retires
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (retire-stg-p (INST-stg i)))
                  (not (equal (INST-stg i) '(commit wbuf0)))
                  (retire-stg-p (INST-stg (step-inst i MT MA MA-sigs)))
                  (not (MT-CMI-p (MT-step MT MA MA-sigs)))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (ISA-mem (ISA-step (INST-pre-ISA i) intr))
                    (ISA-mem (INST-pre-ISA i))))
  :hints (("goal" :cases ((b1p (inst-specultv? i))
                          (b1p (INST-modified? i))))))
)

(encapsulate nil
; This local help lemma shows that retired non-store instructions
; do not affect the value of MT-mem.
; Note. This may be a good example to discuss in a paper, but
; probably difficult to explain in detail.
(local
(defthm MT-mem-MT-step-if-commit-wbuf0-help-help
    (implies (and (inv MT MA)
                  (consp trace)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (no-retired-store-p trace)
                  (no-inst-at-stgs-in-trace '((LSU wbuf0)
                                              (LSU wbuf0 lch)
                                              (complete wbuf0)
                                              (commit wbuf0))
                                            trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (equal (INST-pre-ISA (car trace)) ISA))
             (equal (trace-mem (step-trace trace MT MA sigs
                                           ISA spc smc)
                               (ISA-mem ISA))
                    (ISA-mem ISA)))
  :hints (("goal" :expand (TRACE-MEM
                 (LIST (STEP-INST (CAR TRACE) MT MA SIGS)
                       (FETCHED-INST MT
                                     (ISA-STEP (INST-PRE-ISA (CAR TRACE))
                                               '(ISA-input 0))
                                     (B-IOR (INST-SPECULTV? (CAR TRACE))
                                            (INST-START-SPECULTV? (CAR TRACE)))
```

```
                                (INST-MODIFIED? (CAR TRACE))))
              (ISA-MEM (INST-PRE-ISA (CAR TRACE))))))))

(local
(defthm MT-mem-MT-step-if-commit-wbuf0-help
    (implies (and (inv MT MA)
                  (consp trace)
                  (subtrace-p trace MT)
                  (member-equal i trace) (INST-p i)
                  (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (release-wbuf0? (MA-LSU MA) sigs))
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (equal (INST-stg i) '(commit wbuf0)))
             (equal (trace-mem (step-trace trace MT MA sigs
                                           (INST-pre-ISA (car trace))
                                           spc smc)
                               (ISA-mem (INST-pre-ISA (car trace))))
                    (ISA-mem (INST-post-ISA i))))
  :hints (("goal" :induct t)
          (when-found (INST-LISTP (CDR TRACE))
                      (:cases ((consp (cdr trace))))))))

; If Instruction i at write buffer 0 is released this cycle,
; the ideal memory state is the memory of the post-ISA of i.
(defthm MT-mem-MT-step-if-commit-wbuf0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (release-wbuf0? (MA-LSU MA) sigs))
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (equal (INST-stg i) '(commit wbuf0)))
             (equal (MT-mem (MT-step MT MA sigs))
                    (ISA-mem (INST-post-ISA i))))
  :hints (("goal" :in-theory (enable INST-in MT-mem MT-step)
                  :use (:instance MT-mem-MT-step-if-commit-wbuf0-help
                                  (trace (MT-trace MT)) (spc 0) (smc 0)))
          ("goal'" :cases ((consp (MT-trace MT))))))

)

(encapsulate nil
(local
(defthm local-help
    (implies (and (inv MT MA)
                  (consp trace)
                  (subtrace-p trace MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-listp trace)
                  (member-equal i trace) (INST-p i)
                  (equal (INST-stg i) '(commit wbuf0)))
             (equal (ISA-mem (INST-pre-ISA i))
                    (trace-mem trace (ISA-mem (INST-pre-ISA (car trace))))))
  :hints ((when-found (INST-PRE-ISA (CAR (CDR TRACE)))
                      (:cases ((consp (cdr trace))))))
  :rule-classes nil))

(local
(defthm ISA-mem-INST-pre-ISA-if-INST-at-commit-wbuf0-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
```

```
                (equal (INST-stg i) '(commit wbuf0)))
           (equal (ISA-mem (INST-pre-ISA i))
                  (MT-mem MT)))
  :hints (("goal" :in-theory (e/d (MT-mem INST-in)
                                  (MT-MEM-=-MA-MEM))
                  :cases ((consp (MT-trace MT))))
          ("subgoal 1" :use (:instance local-help (trace (MT-trace MT)))))))

; Before the write instruction is released, the current memory is
;  the same as the memory in the pre-ISA of the instruction.
(defthm ISA-mem-INST-pre-ISA-if-INST-at-commit-wbuf0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (equal (INST-stg i) '(commit wbuf0)))
             (equal (ISA-mem (INST-pre-ISA i))
                    (MA-mem MA))))
)

; The effect of a store instruction.
(defthm ISA-mem-ISA-step-if-INST-store
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (ISA-input-exint intr)))
                  (b1p (INST-store? i))
                  (not (b1p (INST-excpt? i)))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (equal (ISA-mem (ISA-step (INST-pre-ISA i) intr))
                    (write-mem (INST-src-val3 i)
                               (INST-store-addr i)
                               (ISA-mem (INST-pre-ISA i)))))
  :hints (("goal" :in-theory (enable ISA-step-INST-pre-ISA INST-excpt?
                                     INST-data-access-error?
                                     INST-store-error?
                                     INST-store? INST-cntlv INST-opcode
                                     INST-LSU? INST-ld-st?
                                     ISA-st ISA-sti INST-src-val3
                                     INST-STORE-ADDR  INST-rc INST-im
                                     INST-ra INST-rb
                                     decode rdb logbit*
                                     lift-b-ops))))

; A lemma for the proof of MT-mem-if-step-MT-if-release-wbuf0.
; The effect of an instruction in write buffer 0.
(defthm ISA-mem-ISA-step-if-INST-at-commit-wbuf0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (ISA-input-exint intr)))
                  (equal (INST-stg i) '(commit wbuf0))
                  (not (b1p (inst-specultv? i)))
                  (not (b1p (INST-modified? i))))
             (equal (ISA-mem (ISA-step (INST-pre-ISA i) intr))
                    (write-mem (wbuf-val (LSU-wbuf0 (MA-LSU MA)))
                               (wbuf-addr (LSU-wbuf0 (MA-LSU MA)))
                               (MA-mem MA))))
  :hints (("goal" :cases ((b1p (inst-specultv? i))
                          (not (b1p (INST-store? i))))
                  :in-theory (enable LSU-STORE-IF-AT-LSU-WBUF))))
```

```
; A landmark lemma.
; When the content of write buffer 0 is released, the memory
; is updated with the content of wbuf0.
(defthm MT-mem-if-step-MT-if-release-wbuf0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (b1p (release-wbuf0? (MA-LSU MA) sigs)))
             (equal (MT-mem (MT-step MT MA sigs))
                    (write-mem (wbuf-val (LSU-wbuf0 (MA-LSU MA)))
                               (wbuf-addr (LSU-wbuf0 (MA-LSU MA)))
                               (MA-mem MA))))
  :hints (("goal" :in-theory (enable release-wbuf0? release-wbuf0-ready?
                                     INST-exintr-INST-in-if-not-retired
                                     NOT-INST-SPECULTV-INST-IN-IF-COMMITTED
                                     lift-b-ops committed-p)
                  :restrict ((MT-mem-MT-step-if-commit-wbuf0
                              ((i (INST-at-stg '(commit wbuf0) MT)))))
                  :cases ((b1p (INST-modified?
                                (INST-at-stg '(commit wbuf0) MT)))))))

(local
(defthm mem-match-p-preserved-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (not (MT-CMI-p (MT-step MT MA sigs))))
             (equal (MT-mem (MT-step MT MA sigs))
                    (step-mem MA sigs)))
  :hints (("goal" :in-theory (enable step-mem)))))

; Mem-match-p is invariantly preserved.
(defthm mem-match-p-preserved
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (not (MT-CMI-p (MT-step MT MA sigs))))
             (mem-match-p (MT-step MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable mem-match-p MA-step))))
```

D.6.8 misc-inv.lisp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; misc-inv.lisp
; Author  Jun Sawada, University of Texas at Austin
;
;  This book proves miscellaneous invariant properties such as
;  consistent-MA-p, in-order-dispatch-commit-p, correct-speculation-p,
;  no-specultv-commit-p, correct-exintr-p, and misc-inv.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(in-package "ACL2")

(include-book "MA2-lemmas")
(include-book "MAETT-lemmas")

(deflabel begin-misc-inv)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Consistent-MA-p
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Proof of consistent-MA-p for initial states
(defthm consistent-robe-p-if-MA-flushed-help
    (implies (and (ROB-p ROB) (b1p (ROBE-empty-under? idx2 ROB))
                  (integerp idx2) (rob-index-p idx) (< idx idx2))
             (equal (robe-valid? (nth-robe idx ROB)) 0))
  :hints (("goal" :induct (natural-induction idx2)
                  :in-theory (enable ROBE-EMPTY-UNDER? lift-b-ops
                                     ROBE-EMPTY? equal-b1p-converter))))

(defthm consistent-robe-p-if-MA-flushed
    (implies (and (ROB-p ROB) (b1p (ROB-entries-empty? ROB))
                  (b1p (ROB-empty? ROB))
                  (rob-index-p idx))
             (consistent-robe-p (nth-robe idx ROB) idx ROB))
  :hints (("goal" :in-theory (enable consistent-robe-p ROB-entries-empty?
                                     rob-empty? lift-b-ops)))
  :rule-classes
  ((:rewrite :corollary
             (implies (and (ROB-p ROB) (b1p (ROB-entries-empty? ROB))
                           (b1p (ROB-empty? ROB))
                           (rob-index-p idx))
                      (consistent-robe-p (nth idx (ROB-entries ROB))
                                         idx ROB))
             :hints (("goal" :in-theory (enable nth-robe))))))

(defthm consistent-rob-entries-p-if-MA-flushed-help
    (implies (and (ROB-p ROB) (b1p (ROB-empty? ROB))
                  (b1p (ROB-entries-empty? ROB))
                  (integerp idx) (<= 0 idx) (<= idx *rob-size*))
             (consistent-rob-entries-p (nthcdr (- *rob-size* idx)
                                               (ROB-entries ROB))
                                       (rob-index (- *rob-size* idx)) ROB))
  :hints (("goal" :induct (natural-induction idx)
                  :in-theory (enable rob-index-p)))
  :rule-classes nil)

(defthm consistent-rob-entries-p-if-MA-flushed
    (implies (and (ROB-p ROB) (b1p (ROB-empty? ROB))
                  (b1p (ROB-entries-empty? ROB)))
             (consistent-rob-entries-p (ROB-entries ROB) 0 ROB))
  :hints (("goal" :use (:instance consistent-rob-entries-p-if-MA-flushed-help
                                  (idx *rob-size*)))))

(defthm consistent-DQ-cntlv-p-if-MA-flushed
    (implies (and (MA-state-p MA) (b1p (MA-flushed? MA)))
             (consistent-DQ-cntlv-p (MA-DQ MA)))
  :hints (("goal" :in-theory (enable consistent-DQ-cntlv-p lift-b-ops
                                     MA-flushed? DQ-empty?))))

(defthm consistent-ROB-p-if-MA-flushed
    (implies (and (MA-state-p MA) (b1p (MA-flushed? MA)))
             (consistent-ROB-p (MA-ROB MA)))
  :hints (("goal" :in-theory (enable consistent-ROB-p MA-FLUSHED?
                                     ROB-empty? lift-b-ops
                                     consistent-ROB-flg-p))))

(defthm consistent-LSU-p-if-MA-flushed
    (implies (and (MA-state-p MA) (b1p (MA-flushed? MA)))
             (consistent-LSU-p (MA-LSU MA)))
  :hints (("goal" :in-theory (enable consistent-LSU-p
```

```
                                    MA-flushed? lift-b-ops
                                    LSU-empty?))))

(defthm consistent-MA-p-if-MA-flushed
    (implies (and (MA-state-p MA) (b1p (MA-flushed? MA)))
             (consistent-MA-p MA))
  :Hints (("goal" :in-theory (enable consistent-MA-p ))))

;;; Invariant proof
(defthm consistent-cntlv-p-decode
    (implies (and (IFU-p IFU) (MA-state-p MA))
             (consistent-cntlv-p (DE-cntlv (decode-output IFU MA sigs))))
  :hints (("Goal" :in-theory (enable consistent-cntlv-p lift-b-ops
                                    decode-output))))
(defthm consistent-DQ-cntlv-p-preserved-0
    (implies (and (MA-state-p MA)
                  (consistent-DQ-cntlv-p (MA-DQ MA))
                  (b1p (DE-valid? (step-DE0 (MA-DQ MA) MA sigs))))
             (consistent-cntlv-p (DE-cntlv (step-DE0 (MA-DQ MA) MA sigs))))
  :hints (("Goal" :in-theory (enable step-DE0
                                    consistent-DQ-cntlv-p
                                    DE1-out DE2-out DE3-out
                                    lift-b-ops))))

(defthm consistent-DQ-cntlv-p-preserved-1
    (implies (and (MA-state-p MA)
                  (consistent-DQ-cntlv-p (MA-DQ MA))
                  (b1p (DE-valid? (step-DE1 (MA-DQ MA) MA sigs))))
             (consistent-cntlv-p (DE-cntlv (step-DE1 (MA-DQ MA) MA sigs))))
  :hints (("Goal" :in-theory (enable step-DE1
                                    consistent-DQ-cntlv-p
                                    DE1-out DE2-out DE3-out
                                    lift-b-ops))))

(defthm consistent-DQ-cntlv-p-preserved-2
    (implies (and (MA-state-p MA)
                  (consistent-DQ-cntlv-p (MA-DQ MA))
                  (b1p (DE-valid? (step-DE2 (MA-DQ MA) MA sigs))))
             (consistent-cntlv-p (DE-cntlv (step-DE2 (MA-DQ MA) MA sigs))))
  :hints (("Goal" :in-theory (enable step-DE2
                                    consistent-DQ-cntlv-p
                                    DE1-out DE2-out DE3-out
                                    lift-b-ops))))

(defthm consistent-DQ-cntlv-p-preserved-3
    (implies (and (MA-state-p MA)
                  (consistent-DQ-cntlv-p (MA-DQ MA))
                  (b1p (DE-valid? (step-DE3 (MA-DQ MA) MA sigs))))
             (consistent-cntlv-p (DE-cntlv (step-DE3 (MA-DQ MA) MA sigs))))
  :hints (("Goal" :in-theory (enable step-DE3
                                    consistent-DQ-cntlv-p
                                    lift-b-ops))))

(defthm consistent-DQ-cntlv-p-preserved
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA))
             (consistent-DQ-cntlv-p (step-DQ MA sigs)))
  :hints (("goal" :cases ((consistent-DQ-cntlv-p (MA-DQ MA))))
          ("subgoal 2" :in-theory (enable inv consistent-MA-p))
          ("subgoal 1" :in-theory (enable consistent-DQ-cntlv-p
                                          step-DQ))))
```

```
;; (rob-index (+ 1 idx)) is either 0 or (+ 1 idx) depending on whether
;; (+ 1 idx) is wrapped-around.
(local
(defthm rob-index-+-1
    (implies (rob-index-p idx)
             (equal (rob-index (+ 1 idx))
                    (b-if (logbit *ROB-INDEX-SIZE* (+ 1 idx))
                          0 (+ 1 idx))))
  :hints (("goal" :in-theory (enable lift-b-ops rob-index-p rob-index
                                     unsigned-byte-p)
                  :cases ((B1P (LOGBIT 3 (+ 1 IDX)))))
          ("subgoal 1" :use (:instance logbit-0-if-val-lt-expt-2-width
                                     (val (+ 1 idx)) (width 3))))))

; local opening rules.
(local
(defthm rob-flg-step-rob
    (equal (rob-flg (step-rob MA sigs))
           (B-IF
            (FLUSH-ALL? MA SIGS)
            0
            (B-XOR
             (ROB-FLG (MA-ROB MA))
             (B-XOR (B-AND (COMMIT-INST? MA)
                           (LOGBIT *ROB-INDEX-SIZE*
                                   (+ 1 (ROB-HEAD (MA-ROB MA)))))
                    (B-AND (DISPATCH-INST? MA)
                           (LOGBIT *ROB-INDEX-SIZE*
                                   (+ 1 (ROB-TAIL (MA-ROB MA)))))))))
  :hints (("Goal" :in-theory (enable step-rob)))))

(local
(defthm rob-head-step-rob
    (equal (rob-head (step-rob MA sigs))
           (B-IF (FLUSH-ALL? MA SIGS)
                 0
                 (B-IF (COMMIT-INST? MA)
                       (ROB-INDEX (+ 1 (ROB-HEAD (MA-ROB MA))))
                       (ROB-HEAD (MA-ROB MA)))))
  :hints (("Goal" :in-theory (enable step-rob)))))

(local
(defthm rob-tail-step-rob
    (equal (rob-tail (step-rob MA sigs))
           (B-IF (FLUSH-ALL? MA SIGS)
                 0
                 (B-IF (DISPATCH-INST? MA)
                       (ROB-INDEX (+ 1 (ROB-TAIL (MA-ROB MA))))
                       (ROB-TAIL (MA-ROB MA)))))
  :hints (("Goal" :in-theory (enable step-rob)))))

; Linear lemmas to restrict the range of ROB-tail
(defthm rob-tail-range
    (implies (ROB-p ROB)
             (and (< (rob-tail ROB) 8)
                  (<=  0 (rob-tail ROB))))
  :rule-classes
  ((:linear :corollary
            (implies (ROB-p ROB) (< (rob-tail ROB) 8)))
   (:linear :corollary
            (implies (ROB-p ROB) (<=  0 (rob-tail ROB))))))
```

```
; Linear lemmas to restrict the range of ROB-head
(defthm rob-head-range
    (implies (ROB-p ROB)
             (and (< (rob-head ROB) 8)
                  (<=  0 (rob-head ROB))))
  :rule-classes
  ((:linear :corollary
            (implies (ROB-p ROB) (< (rob-head ROB) 8)))
   (:linear :corollary
            (implies (ROB-p ROB) (<=  0 (rob-head ROB))))))

; If the reorder-buffer is full, no instruction is dispatched.
; Unfortunately, the version with ROB-empty? does not work in the
; proof for consistent-robe-p-preserved.
(defthm not-dispatch-INST-if-rob-is-full
    (implies (and (MA-state-p MA)
                  (b1p (ROB-flg (MA-rob MA)))
                  (equal (ROB-head (MA-ROB MA)) (ROB-tail (MA-ROB MA))))
             (equal (dispatch-INST? MA) 0))
  :hints (("goal" :in-theory (enable MA-def bv-eqv-iff-equal lift-b-ops
                                     equal-b1p-converter))))

; If the reorder-buffer is empty, no instruction is committed.
; Unfortunately, the version with ROB-full? does not work in the
; proof for consistent-robe-p-preserved.
(defthm not-commit-INST-if-rob-is-empty
    (implies (and (MA-state-p MA)
                  (consistent-rob-p (MA-ROB MA))
                  (not (b1p (ROB-flg (MA-rob MA))))
                  (equal (ROB-head (MA-ROB MA)) (ROB-tail (MA-ROB MA))))
             (equal (commit-INST? MA) 0))
  :hints (("goal" :in-theory (enable MA-def bv-eqv-iff-equal lift-b-ops
                                     equal-b1p-converter))))

(encapsulate nil
(local
(defthm cntlv-sync-branch-of-dispatch-cntlv-exclusive-help-help
    (implies (and (consistent-cntlv-p (DE-cntlv (DQ-DE0 (MA-DQ MA))))
                  (b1p (de-valid? (DQ-DE0 (MA-DQ MA))))
                  (b1p (cntlv-sync? (dispatch-cntlv MA))))
             (equal (logbit 3 (cntlv-exunit (dispatch-cntlv MA))) 0))
  :hints (("goal" :in-theory (enable dispatch-cntlv inv
                                     consistent-MA-p consistent-DQ-cntlv-p
                                     CONSISTENT-CNTLV-P lift-b-ops
                                     equal-b1p-converter)))))

(local
(defthm cntlv-sync-cntlv-wb-of-dispatch-cntlv-exclusive-help-help
    (implies (and (consistent-cntlv-p (DE-cntlv (DQ-DE0 (MA-DQ MA))))
                  (b1p (de-valid? (DQ-DE0 (MA-DQ MA))))
                  (b1p (cntlv-sync? (dispatch-cntlv MA))))
             (equal (cntlv-wb? (dispatch-cntlv MA)) 0))
    :hints (("goal" :in-theory (enable dispatch-cntlv inv
                                     consistent-MA-p consistent-DQ-cntlv-p
                                     CONSISTENT-CNTLV-P lift-b-ops
                                     equal-b1p-converter)))))

(local
(defthm cntlv-sync-branch-of-dispatch-cntlv-exclusive-help
    (implies (and (inv MT MA)
                  (b1p (de-valid? (DQ-DE0 (MA-DQ MA))))
                  (b1p (cntlv-sync? (dispatch-cntlv MA))))
```

```
            (equal (logbit 3 (cntlv-exunit (dispatch-cntlv MA))) 0))
  :hints (("goal" :in-theory (enable inv
                                     consistent-MA-p consistent-DQ-cntlv-p
                                     lift-b-ops)))))

(local
(defthm cntlv-sync-cntlv-wb-of-dispatch-cntlv-exclusive-help
    (implies (and (inv MT MA)
                  (b1p (de-valid? (DQ-DE0 (MA-DQ MA))))
                  (b1p (cntlv-sync? (dispatch-cntlv MA))))
             (equal (cntlv-wb? (dispatch-cntlv MA)) 0))
    :hints (("goal" :in-theory (enable inv
                                     consistent-MA-p consistent-DQ-cntlv-p
                                     lift-b-ops)))))

(local
 (defthm DQ-de0-valid-if-dispatch-inst
     (implies (and (MA-state-p MA) (b1p (dispatch-inst? MA)))
              (equal (de-valid? (DQ-DE0 (MA-DQ MA))) 1))
   :hints (("goal" :in-theory (enable MA-def lift-b-ops
                                      equal-b1p-converter)))))

(defthm cntlv-sync-branch-of-dispatch-cntlv-exclusive
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (b1p (cntlv-sync? (dispatch-cntlv MA))))
             (equal (logbit 3 (cntlv-exunit (dispatch-cntlv MA))) 0)))

(defthm cntlv-sync-cntlv-wb-of-dispatch-cntlv-exclusive
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (b1p (cntlv-sync? (dispatch-cntlv MA))))
             (equal (cntlv-wb? (dispatch-cntlv MA)) 0)))
)

(encapsulate nil
(local
(defthm rob-tail-7-if-carries-out
    (implies (and (ROB-p ROB)
                  (b1p (logbit 3 (+ 1 (rob-tail ROB)))))
             (equal (rob-tail ROB) 7))
  :Hints (("goal" :use (:instance logbit-0-if-val-lt-expt-2-width
                                  (val (+ 1 (rob-tail ROB))) (width 3))
                  :in-theory (enable rob-p rob-index-p unsigned-byte-p)))))

(local
(defthm rob-head-7-if-carries-out
    (implies (and (ROB-p ROB) (b1p (logbit 3 (+ 1 (rob-head ROB)))))
             (equal (rob-head ROB) 7))
  :Hints (("goal" :use (:instance logbit-0-if-val-lt-expt-2-width
                                  (val (+ 1 (rob-head ROB))) (width 3))
                  :in-theory (enable rob-p rob-index-p unsigned-byte-p)))))

(local
(defun consistent-robe-p-1 (robe idx ROB)
   (equal (robe-valid? robe)
          (if (b1p (ROB-flg ROB))
              (if (or (<= (ROB-head ROB) idx) (< idx (ROB-tail ROB))) 1 0)
              (if (and (<= (ROB-head ROB) idx) (< idx (ROB-tail ROB))) 1 0)))))
```

```
(local
(defun consistent-robe-p-2 (robe)
    (implies (b1p (robe-valid? robe))
             (and (not (b1p (b-and (robe-branch? robe) (robe-sync? robe))))
                  (not (b1p (b-and (robe-wb? robe) (robe-sync? robe))))))))

(local
(defthm consistent-robe-p*
    (equal (consistent-robe-p robe idx ROB)
           (and (consistent-robe-p-1 robe idx ROB)
                (consistent-robe-p-2 robe)))
  :hints (("goal" :in-theory (enable consistent-robe-p)))
  :rule-classes :definition))

(local
(in-theory (disable consistent-robe-p-1 consistent-robe-p-2
                    consistent-robe-p*)))

; consistent-robe-p is preserved.
(local
(defthm consistent-robe-p-1-preserved
    (implies (and (rob-index-p idx)
                  (MA-state-p MA)
                  (MAETT-p MT)
                  (consistent-ROB-p (MA-ROB MA))
                  (consistent-robe-p robe idx (MA-ROB MA)))
             (consistent-robe-p-1 (step-robe robe idx (MA-ROB MA) MA sigs)
                                  idx  (step-rob MA sigs)))
  :hints (("goal" :in-theory (enable consistent-robe-p-1
                                     consistent-robe-p*
                                     step-robe
                                     consistent-rob-p
                                     rob-index-p
                                     unsigned-byte-p
                                     consistent-rob-flg-p
                                     lift-b-ops
                                     bv-eqv-iff-equal
                                     ROBE-RECEIVE-INST?
                                     )))))

(local
(defthm consistent-robe-p-2-preserved
    (implies (and (inv MT MA)
                  (rob-index-p idx)
                  (MA-state-p MA)
                  (MAETT-p MT)
                  (consistent-robe-p robe idx (MA-ROB MA)))
             (consistent-robe-p-2 (step-robe robe idx (MA-ROB MA) MA sigs)))
  :hints (("goal" :in-theory (enable consistent-robe-p-2
                                     CONSISTENT-ROBE-P*
                                     ROBE-RECEIVE-INST?
                                     lift-b-ops
                                     consistent-cntlv-p
                                     step-robe)))))

(defthm consistent-robe-p-preserved
    (implies (and (inv MT MA)
                  (rob-index-p idx)
                  (MA-state-p MA)
                  (MAETT-p MT)
                  (consistent-ROB-p (MA-ROB MA))
                  (consistent-robe-p robe idx (MA-ROB MA)))
```

```
            (consistent-robe-p (step-robe robe idx (MA-ROB MA) MA sigs)
                               idx  (step-rob MA sigs)))
  :hints (("goal" :in-theory (enable consistent-robe-p*))))

(defthm consistent-ROB-entries-p-preserved
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (rob-index-p idx)
                  (consistent-ROB-p (MA-ROB MA))
                  (consistent-ROB-entries-p robe-list idx (MA-ROB MA)))
             (consistent-ROB-entries-p (step-robe-list robe-list idx
                                                       (MA-rob MA) MA sigs)
                                       idx
                                       (step-rob MA sigs)))
  :hints (("Goal" :in-theory (enable STEP-ROBE-LIST))))

; consistent-rob-flg-p is preserved.
(defthm consistent-rob-flg-p-preserved
    (implies (and (MA-state-p MA)
                  (consistent-rob-flg-p (MA-ROB MA))
                  (consistent-rob-p (MA-rob MA)))
             (consistent-ROB-flg-p (step-ROB MA sigs)))
  :hints (("goal" :in-theory (enable consistent-rob-flg-p lift-b-ops
                                     consistent-rob-p))))
)

(local
(defthm rob-entries-step-rob
    (equal (rob-entries (step-rob MA sigs))
           (step-rob-entries (rob-entries (MA-rob MA)) (MA-rob MA) MA sigs))
  :hints (("Goal" :in-theory (enable step-rob)))))

; consistent-ROB-p is preserved.
(defthm consistent-ROB-p-preserved
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA))
             (consistent-ROB-p (step-rob MA sigs)))
  :hints (("goal" :cases  ((consistent-ROB-p (MA-ROB MA))))
          ("subgoal 2" :in-theory (enable inv consistent-MA-p))
          ("subgoal 1" :in-theory (enable consistent-rob-p step-rob-entries))))

;; Proof of consistent-LSU-p-preserved
(defthm wbuf0-valid-step-wbuf0-if-rbuf-wbuf0
    (implies (and (inv MT MA)
                  (b1p (rbuf-valid? (step-rbuf (MA-LSU MA) MA sigs)))
                  (b1p (rbuf-wbuf0? (step-rbuf (MA-LSU MA) MA sigs)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (equal (wbuf-valid? (step-wbuf0 (MA-LSU MA) MA sigs)) 1))
  :hints (("Goal" :in-theory (enable step-rbuf step-wbuf0
                                     wbuf1-output update-wbuf0
                                     RELEASE-WBUF0?
                                     issued-write
                                     lift-b-ops equal-b1p-converter))))

(defthm wbuf1-valid-step-wbuf1-if-rbuf-wbuf1
    (implies (and (inv MT MA)
                  (b1p (rbuf-valid? (step-rbuf (MA-LSU MA) MA sigs)))
                  (b1p (rbuf-wbuf1? (step-rbuf (MA-LSU MA) MA sigs)))
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (equal (wbuf-valid? (step-wbuf1 (MA-LSU MA) MA sigs)) 1))
  :hints (("Goal" :in-theory (enable step-rbuf step-wbuf1 lift-b-ops
                                     issued-write update-wbuf1
                                     RELEASE-WBUF0?
```

```
                                    equal-b1p-converter))))

(defthm uniq-inst-at-stg-non-commit-wbuf0
     (implies (and (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (not (b1p (wbuf-commit? (LSU-wbuf0 (MA-LSU MA)))))
                   (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
              (uniq-inst-at-stgs '((LSU wbuf0)
                                   (LSU wbuf0 lch)
                                   (complete wbuf0))
                                 MT))
  :hints (("Goal" :use ((:instance UNIQ-INST-AT-LSU-WBUF0-IF-VALID)
                        (:instance uniq-inst-at-stgs-remove-equal
                                   (stgs '((LSU wbuf0)
                                           (LSU wbuf0 lch)
                                           (complete wbuf0)
                                           (commit wbuf0)))
                                   (stg '(commit wbuf0)))))))

(defthm not-committed-p-inst-at-non-commit-wbuf0
    (implies (and (inv MT MA)
                  (uniq-inst-at-stgs '((LSU wbuf0)
                                       (LSU wbuf0 lch)
                                       (complete wbuf0))
                                     MT)
                  (MAETT-p MT) (MA-state-p MA))
             (not (committed-p (inst-at-stgs '((LSU wbuf0)
                                               (LSU wbuf0 lch)
                                               (complete wbuf0))
                                             MT))))
  :hints (("Goal" :use ((:instance uniq-inst-at-stgs*
                                   (stgs '((LSU wbuf0)
                                           (LSU wbuf0 lch)
                                           (complete wbuf0))))
                        (:instance uniq-inst-at-stgs*
                                   (stgs '((LSU wbuf0 lch)
                                           (complete wbuf0))))
                        (:instance inst-at-stgs*
                                   (stgs '((LSU wbuf0)
                                           (LSU wbuf0 lch)
                                           (complete wbuf0))))
                        (:instance inst-at-stgs*
                                   (stgs '((LSU wbuf0 lch)
                                           (complete wbuf0)))))))))

(defthm wbuf0-commit-if-wbuf1-commit
    (implies (and (inv MT MA)
                  (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                  (b1p (wbuf-commit? (LSU-wbuf1 (MA-LSU MA))))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (wbuf-commit? (LSU-wbuf0 (MA-LSU MA))) 1))
  :hints (("Goal" :use (:instance INST-in-order-p-wbuf0-wbuf1
                                  (i (INST-at-stgs '((LSU wbuf0)
                                                     (LSU wbuf0 lch)
                                                     (complete wbuf0)) MT))
                                  (j (INST-at-stg '(commit wbuf1) MT)))
                  :in-theory (enable lift-b-ops equal-b1p-converter))))

(defthm wbuf0-valid-step-wbuf0
    (implies (and (inv MT MA)
                  (b1p (wbuf-valid? (step-wbuf1 (MA-LSU MA) MA sigs)))
                  (MAETT-p MT) (MA-state-p MA))
```

```
            (b1p (wbuf-valid? (step-wbuf0 (MA-LSU MA) MA sigs))))
  :Hints (("Goal" :in-theory (enable step-wbuf1 step-wbuf0 lift-b-ops
                                     wbuf1-output update-wbuf1
                                     ISSUED-WRITE
                                     update-wbuf0))))

(defthm consistent-LSU-p-preserved
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (consistent-LSU-p (step-LSU MA sigs)))
  :hints (("Goal" :in-theory (enable consistent-LSU-p step-LSU lift-b-ops))))

; consistent-MA-p is preserved.
(defthm consistent-MA-p-preserved
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (consistent-MA-p (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable consistent-MA-p))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Proof about in-order-dispatch-commit-p
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;; Proof of in-order-dispatch-commit-p for initial states
(defthm in-order-dispatch-commit-p-init-MT
    (in-order-dispatch-commit-p (init-MT MA))
  :hints (("goal" :in-theory (enable init-MT in-order-dispatch-commit-p))))

;;;;; Invariant proof
(defthm step-trace-cdr-if-step-INST-car-is-IFU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (not (b1p (INST-exintr-now? (car trace) MA sigs)))
                  (IFU-stg-p (INST-stg (step-INST (car trace) MT MA sigs))))
             (equal (step-trace (cdr trace) MT MA sigs ISA spc smc)
                    nil))
  :hints (("goal" :use (:instance INST-is-at-one-of-the-stages
                                  (i (car trace)))
                  :in-theory (disable INST-is-at-one-of-the-stages)
                  :do-not-induct t)
          ("subgoal 2" :cases ((consp (cdr trace))))))

(encapsulate nil
(local
(defthm INST-in-order-car-car-tail-p-help
    (implies (and (consp sub)
                  (tail-p trace sup)
                  (tail-p sub trace)
                  (not (equal sub trace)))
             (member-in-order (car trace) (car sub) sup))
  :hints (("goal" :In-theory (enable member-in-order* tail-p)))))

(defthm INST-in-order-car-car-tail-p
    (implies (and (consp sub)
                  (subtrace-p trace MT)
                  (tail-p sub trace)
                  (not (equal sub trace)))
             (INST-in-order-p (car trace) (car sub) MT))
  :hints (("goal" :In-theory (enable INST-in-order-p subtrace-p))))
)
```

```
(encapsulate nil
(local
(defthm not-member-in-order-if-i-is-IFU
    (implies (and (in-order-trace-p trace)
                  (IFU-stg-p (INST-stg i)))
             (not (member-in-order I J trace)))
  :hints (("goal" :in-theory (enable member-in-order*)))))

(defthm not-INST-in-order-if-i-is-IFU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (IFU-stg-p (INST-stg i)))
             (not (INST-in-order-p I J MT)))
  :Hints (("goal" :in-theory (enable INST-in-order-p inv
                                     in-order-dispatch-commit-p))))
)

(defthm DQ-stg-idx-non-negative
    (implies (and (INST-p i) (DQ-stg-p (INST-stg i)))
             (<= 0 (DQ-stg-idx (INST-stg i))))
  :hints (("goal" :in-theory (enable INST-p stage-p DQ-stg-p)))
  :rule-classes :linear)

(defthm DQ-stg-p-decremented
    (implies (and (INST-p i)
                  (DQ-stg-p (INST-stg i))
                  (< 0 (DQ-stg-IDX (INST-stg i))))
             (DQ-stg-p (list 'DQ (+ -1 (DQ-stg-IDX (INST-stg i))))))
  :Hints (("goal" :in-theory (enable DQ-stg-p INST-p stage-p))))

(encapsulate nil
(local
 (defthm DQ-stg-idx-monotone-help-help
     (implies (and (consp trace)
                   (INST-listp trace)
                   (INST-p j)
                   (DQ-stg-p (INST-stg (car trace)))
                   (DQ-stg-p (INST-stg j))
                   (in-order-trace-p trace)
                   (member-equal j (cdr trace))
                   (in-order-DQ-trace-p (cdr trace)
                                        (+ 1 (DQ-stg-idx (INST-stg (car trace))))))
              (< (DQ-stg-idx (INST-stg (car trace)))
                 (DQ-stg-idx (INST-stg j))))
   :hints (("goal" :in-theory (enable dispatched-p)))))

(local
(defthm DQ-stg-idx-monotone-help
    (implies (and (in-order-trace-p trace)
                  (in-order-DQ-trace-p trace idx)
                  (INST-p i)
                  (INST-listp trace)
                  (member-in-order i j trace)
                  (DQ-stg-p (INST-stg i))
                  (DQ-stg-p (INST-stg j)))
             (< (DQ-stg-idx (INST-stg i)) (DQ-stg-idx (INST-stg j))))
  :Hints (("Goal" :in-theory (enable member-in-order*)))))

(defthm DQ-stg-idx-monotone
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in-order-p i j MT)
```

```
                (INST-p i) (INST-p j)
                (DQ-stg-p (INST-stg i))
                (DQ-stg-p (INST-stg j)))
           (< (DQ-stg-idx (INST-stg i)) (DQ-stg-idx (INST-stg j))))
  :hints (("goal" :in-theory (enable inv in-order-DQ-p
                                     in-order-dispatch-commit-p
                                     INST-in-order-p)))
  :rule-classes nil)
)

(encapsulate nil
(local
 (defthm not-conflict-inst-exintr-now
     (implies (and (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (INST-p j)
                   (B1P (INST-EXINTR-NOW? i MA SIGS))
                   (not (b1p (INST-exintr-now? j MA sigs))))
              (not (DQ-stg-p (INST-stg (step-inst j MT MA sigs)))))
   :hints (("goal" :cases ((b1p (ex-intr? MA sigs))))
           ("subgoal 1" :use (:instance INST-is-at-one-of-the-stages (i j))
                        :in-theory (disable INST-is-at-one-of-the-stages)))))

(local
 (defthm ex-intr-inst-exintr-now-DQ-stg-conflict
     (implies (and (INST-p i)
                   (b1p (ex-intr? MA sigs))
                   (not (b1p (INST-exintr-now? i MA sigs))))
              (not (DQ-stg-p (INST-stg (step-INST i MT MA sigs)))))
   :hints (("goal"  :use (:instance INST-is-at-one-of-the-stages)
                    :in-theory (disable INST-is-at-one-of-the-stages)))))



(defthm DQ-stg-p-second-if-first-inst-is-DQ
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-p j)
                  (INST-in-order-p i j MT)
                  (DQ-stg-p (INST-stg i))
                  (DQ-stg-p (INST-stg j)))
             (DQ-stg-p (INST-stg (step-INST j MT MA sigs))))
  :hints (("goal" :use (:instance DQ-stg-idx-monotone)
                  :in-theory (enable step-INST-opener step-INST-dq))))

(local
(defthm IFU-or-DQ-stg-step-INST-if-INST-in-order
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (DQ-stg-p (INST-stg (step-INST i MT MA sigs)))
                  (INST-p i) (INST-p j)
                  (INST-in i MT) (INST-in j MT)
                  (INST-in-order-p i j MT)
                  (not  (IFU-stg-p (INST-stg (step-INST j MT MA sigs)))))
             (DQ-stg-p (INST-stg (step-INST j MT MA sigs))))
  :hints (("goal" :use ((:instance INST-is-at-one-of-the-stages)
                        (:instance INST-is-at-one-of-the-stages (i j)))
                  :in-theory (disable INST-is-at-one-of-the-stages)))))

(local
(defthm no-dispatch-inst-step-trace-cdr-if-step-INST-car-is-DQ-help
    (implies (and (inv MT MA)
```

```
                (MAETT-p MT) (MA-state-p MA)
                (MA-input-p sigs)
                (subtrace-p trace MT)
                (not (equal sub trace))
                (tail-p sub trace)
                (INST-listp sub)
                (INST-listp trace)
                (not (b1p (inst-exintr-now? (car trace) MA sigs)))
                (DQ-stg-p (INST-stg (step-INST (car trace) MT MA sigs))))
           (no-dispatched-inst-p
            (step-trace sub MT MA sigs ISA spc smc)))
  :hints (("goal" :induct (step-trace sub MT MA sigs ISA spc smc)
                  :restrict ((IFU-or-DQ-stg-step-INST-if-INST-in-order
                              ((i (car trace)))))
                  :in-theory (enable dispatched-p*))
          (when-found (DQ-STG-P (INST-STG (STEP-INST (CAR SUB) MT MA SIGS)))
                      (:cases ((consp trace))))
          (when-found (IFU-STG-P (INST-STG (STEP-INST (CAR SUB) MT MA SIGS)))
                      (:cases ((consp trace)))))))

; The instructions are not dispatched in this cycle, if a preceding
; instruction is in the DQ stage in the next cycle.
(defthm no-dispatch-inst-step-trace-cdr-if-step-INST-car-is-DQ
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (not (b1p (inst-exintr-now? (car trace) MA sigs)))
                  (DQ-stg-p (INST-stg (step-INST (car trace) MT MA sigs))))
             (no-dispatched-inst-p
              (step-trace (cdr trace) MT MA sigs ISA spc smc))))
)

(local
(encapsulate nil
; If instruction i is in the execute stage next cycle, a subsequent
; instruction j is not committed in the next cycle.
(local
(defthm not-commit-if-earlier-inst-in-execute-stg
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in-order-p i j MT)
                  (INST-p i) (INST-in i MT)
                  (INST-p j) (INST-in j MT)
                  (execute-stg-p (INST-stg (step-INST i MT MA sigs))))
             (not (commit-stg-p (INST-stg (step-INST j MT MA sigs)))))
  :hints (("goal" :use ((:instance INST-is-at-one-of-the-stages)
                        (:instance INST-is-at-one-of-the-stages (i j)))
                  :in-theory (disable INST-is-at-one-of-the-stages)))))

(local
(defthm not-retire-if-earlier-inst-in-execute-stg
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in-order-p i j MT)
                  (INST-p i) (INST-in i MT)
                  (INST-p j) (INST-in j MT)
                  (execute-stg-p (INST-stg (step-INST i MT MA sigs))))
             (not (retire-stg-p (INST-stg (step-INST j MT MA sigs)))))
  :hints (("goal" :use ((:instance INST-is-at-one-of-the-stages)
                        (:instance INST-is-at-one-of-the-stages (i j)))
```

```
                :in-theory (disable INST-is-at-one-of-the-stages)))))

; Local lemmas
; If external exception occurs this cycle, no instruction will be
; in the execute stage in the next cycle.
(local
(defthm  not-execute-stg-p-step-INST-if-ex-intr
     (implies (and (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (B1P (ex-intr? MA sigs))
                   (INST-p i) (INST-in i MT)
                   (not (B1P (INST-EXINTR-NOW? i MA SIGS))))
              (not (execute-stg-p (INST-stg (step-INST i MT MA sigs)))))
   :hints (("goal" :use (:instance INST-is-at-one-of-the-stages)
                   :in-theory (disable INST-is-at-one-of-the-stages)))))

(local
(defthm  not-execute-stg-p-step-INST-if-INST-exintr-now?
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (not (B1P (INST-EXINTR-NOW? i MA SIGS)))
                  (B1P (INST-EXINTR-NOW? j MA SIGS)))
             (not (execute-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("goal" :use (:instance INST-is-at-one-of-the-stages)
                  :in-theory (disable INST-is-at-one-of-the-stages))
          (when-found (DQ-STG-P (INST-STG I))
                      (:cases ((b1p (ex-intr? MA sigs))))))))

(local
(defthm no-complete-inst-step-trace-cdr-if-step-INST-car-execute-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (not (equal sub trace))
                  (tail-p sub trace)
                  (INST-listp sub)
                  (INST-listp trace)
                  (execute-stg-p (INST-stg (step-INST (car trace) MT MA sigs)))
                  (not (b1p (INST-exintr-now? (car trace) MA sigs))))
             (no-commit-inst-p
              (step-trace sub MT MA sigs ISA spc smc)))
  :hints (("goal" :induct (step-trace sub MT MA sigs ISA spc smc)
                  :restrict ((not-commit-if-earlier-inst-in-execute-stg
                              ((i (car trace))))
                             (not-retire-if-earlier-inst-in-execute-stg
                              ((i (car trace)))))
                  :expand (INST-listp trace)))))

; A local lemma to help the proof of in-order-trace-p-step-trace.
; If instruction (car trace) is in the execute stage in the next cycle,
; no instruction in (cdr trace) will be committed.  This is
; practically the proof of in-order commit.
(defthm no-complete-inst-step-trace-cdr-if-step-INST-car-execute
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (not (b1p (INST-exintr-now? (car trace) MA sigs)))
                  (execute-stg-p (INST-stg (step-INST (car trace)
                                                      MT MA sigs))))
             (no-commit-inst-p
```

```
             (step-trace (cdr trace) MT MA sigs ISA spc smc))))
))

; A local lemma to help the proof of in-order-trace-p-step-trace.
; If instruction (car trace) is in the complete stage in the next cycle,
; no instruction in (cdr trace) will be committed.  This is
; practically the proof of in-order commit.
(encapsulate nil
; If instruction i is in the complete stage next cycle, a subsequent
; instruction j is not committed in the next cycle.
(local
(defthm not-commit-if-earlier-inst-in-complete-stg
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in-order-p i j MT)
                  (INST-p i) (INST-in i MT)
                  (INST-p j) (INST-in j MT)
                  (complete-stg-p (INST-stg (step-INST i MT MA sigs))))
             (not (commit-stg-p (INST-stg (step-INST j MT MA sigs)))))
  :hints (("goal" :use ((:instance INST-is-at-one-of-the-stages)
                        (:instance INST-is-at-one-of-the-stages (i j)))
                  :in-theory (disable INST-is-at-one-of-the-stages)))))

(local
(defthm not-retire-if-earlier-inst-in-complete-stg
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in-order-p i j MT)
                  (INST-p i) (INST-in i MT)
                  (INST-p j) (INST-in j MT)
                  (complete-stg-p (INST-stg (step-INST i MT MA sigs))))
             (not (retire-stg-p (INST-stg (step-INST j MT MA sigs)))))
  :hints (("goal" :use ((:instance INST-is-at-one-of-the-stages)
                        (:instance INST-is-at-one-of-the-stages (i j)))
                  :in-theory (disable INST-is-at-one-of-the-stages)))))

(local
(defthm  not-complete-stg-p-step-INST-if-ex-intr
     (implies (and (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (B1P (ex-intr? MA sigs))
                   (INST-p i) (INST-in i MT)
                   (not (B1P (INST-EXINTR-NOW? i MA SIGS))))
              (not (complete-stg-p (INST-stg (step-INST i MT MA sigs)))))
   :hints (("goal" :use (:instance INST-is-at-one-of-the-stages)
                   :in-theory (disable INST-is-at-one-of-the-stages)))))

(local
(defthm  not-complete-stg-p-step-INST-if-INST-exintr-now?
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-p i) (INST-in i MT)
                  (not (B1P (INST-EXINTR-NOW? i MA SIGS)))
                  (B1P (INST-EXINTR-NOW? j MA SIGS)))
             (not (complete-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("goal" :use (:instance INST-is-at-one-of-the-stages)
                  :in-theory (disable INST-is-at-one-of-the-stages))
          (when-found (DQ-STG-P (INST-STG I))
                      (:cases ((b1p (ex-intr? MA sigs))))))))

(local
(defthm no-complete-inst-step-trace-cdr-if-step-INST-car-complete-help
```

```
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (not (equal sub trace))
                  (tail-p sub trace)
                  (INST-listp sub) (INST-listp trace)
                  (complete-stg-p
                   (INST-stg (step-INST (car trace) MT MA sigs)))
                  (not (b1p (INST-exintr-now? (car trace) MA sigs))))
             (no-commit-inst-p
              (step-trace sub MT MA sigs ISA spc smc)))
  :hints (("goal" :induct (step-trace sub MT MA sigs ISA spc smc)
                  :restrict ((NOT-COMMIT-IF-EARLIER-INST-IN-COMPLETE-STG
                              ((i (car trace))))
                             (NOT-RETIRE-IF-EARLIER-INST-IN-COMPLETE-STG
                              ((i (car trace)))))
                  :expand (INST-listp trace)))))

(defthm no-complete-inst-step-trace-cdr-if-step-INST-car-complete
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (complete-stg-p
                   (INST-stg (step-INST (car trace) MT MA sigs)))
                  (not (b1p (INST-exintr-now? (car trace) MA sigs))))
             (no-commit-inst-p
              (step-trace (cdr trace) MT MA sigs ISA spc smc))))
)

(defthm in-order-trace-p-step-trace
    (implies (and (inv MT MA)
                  (MA-state-p MA) (MAETT-p MT) (MA-input-p sigs)
                  (INST-listp trace)
                  (subtrace-p trace MT))
             (in-order-trace-p (step-trace trace MT MA sigs ISA smc spc)))
  :hints (("goal" :in-theory (enable dispatched-p* committed-p*))))

(defthm in-order-dispatch-commit-p-preserved
    (implies (and (inv MT MA) (MA-state-p MA) (MAETT-p MT) (MA-input-p sigs))
             (in-order-dispatch-commit-p (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable MT-step in-order-dispatch-commit-p))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Proof of correct-speculation-p-preserved
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; correct-speculation-p is true for init-MAETT.
(defthm correct-speculation-p-init-MAETT
    (implies (MA-state-p MA)
             (correct-speculation-p (init-MT MA)))
  :hints (("Goal" :in-theory (enable init-MT correct-speculation-p))))

(defthm inst-wrong-branch-step-INST-if-not-retire-after-step
     (implies (and (INST-p i) (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                   (complete-stg-p (INST-stg i))
                   (not (retire-stg-p (INST-stg (step-INST i MT MA sigs)))))
              (equal (inst-wrong-branch? (step-INST i MT MA sigs))
                     (inst-wrong-branch? i)))
  :hints (("goal" :in-theory (enable inst-wrong-branch? lift-b-ops))))

; The machine may speculatively execute instructions following a
; certain type of instructions.  These instructions initiating
```

```
; speculative execution can be characterized with
; INST-start-specultv?. (INST-start-specultv? i) does not change its
; value unless i retires or advances from IFU stage.
(defthm inst-start-specultv-step-complete-INST
     (implies (and (inv MT MA)
                   (MAETT-p MT) (MA-state-p MA)
                   (MA-input-p sigs)
                   (INST-p i)
                   (INST-in i MT)
                   (complete-stg-p (INST-stg i))
                   (not (b1p (INST-specultv? i)))
                   (not (b1p (INST-modified? i)))
                   (not (b1p (INST-cause-jmp? i MT MA sigs))))
              (equal (inst-start-specultv? (step-INST i MT MA sigs))
                     (inst-start-specultv? i)))
  :hints (("goal" :in-theory (e/d (INST-start-specultv? lift-b-ops
                                     equal-b1p-converter)
                                  (inst-is-at-one-of-the-stages))
                  :use (:instance INST-is-at-one-of-the-stages))
          (when-found (INST-WRONG-BRANCH? (STEP-INST I MT MA SIGS))
              (:cases
               ((complete-stg-p (INST-stg (step-INST i MT MA sigs))))))))

; If instruction i is a correctly predicted branch instruction in the IFU,
; and if fetch-inst? is asserted, i is still correctly predicted
; branch in DQ.  In the IFU stage, we all assume that a branch is not
; taken, and when it is decoded, a branch prediction mechanism
; predicts branch outcome.  If branch i is predicted to be taken, then a
; new instruction won't be fetched during this cycle, i.e. fetch-inst?
; is not asserted.  If the prediction mechanism guesses that i is not
; taken, we continue fetch instruction and fetch-inst? is asserted.
(defthm inst-wrong-branch-step-IFU-inst-if-fetch-inst
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (INST-p I)
                  (INST-in i MT)
                  (b1p (fetch-inst? MA sigs))
                  (IFU-stg-p (INST-stg i)))
             (equal (INST-wrong-branch? (step-INST i MT MA sigs))
                    (INST-wrong-branch? i)))
  :hints (("Goal" :in-theory (enable INST-wrong-branch? lift-b-ops
                                     fetch-inst?
                                     equal-b1p-converter
                                     equal-to-b1p-b-eqv))))

(defthm not-inst-start-specultv-step-inst-if-fetch-inst
    (implies (and (inv MT MA)
                  (MA-state-p MA) (MAETT-p MT)
                  (MA-input-p sigs) (INST-p i)
                  (INST-in i MT)
                  (IFU-stg-p (INST-stg i))
                  (b1p (fetch-inst? MA sigs)))
             (equal (INST-start-specultv? (step-INST i MT MA sigs))
                    (INST-start-specultv? i)))
  :hints (("Goal" :in-theory (enable inst-start-specultv?
                                     lift-b-ops
                                     equal-b1p-converter))))

(encapsulate nil
(local
 (defthm not-commit-stg-p-step-inst-if-earlier-is-committed-p
     (implies (and (inv MT MA)
```

```
                  (MAETT-p MT) (MA-state-p MA)
                  (inst-in-order-p i j MT)
                  (INST-p i) (INST-in i MT)
                  (INST-p j) (INST-in j MT)
                  (not (committed-p i)))
             (not (commit-stg-p (INST-stg (step-INST j MT MA sigs)))))
   :hints (("goal" :use ((:instance INST-is-at-one-of-the-stages)
                         (:instance INST-is-at-one-of-the-stages (i j)))
                  :in-theory (disable INST-is-at-one-of-the-stages)))))

(local
 (defthm not-retire-stg-p-step-inst-if-earlier-is-committed-p
     (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (inst-in-order-p i j MT)
                  (INST-p i) (INST-in i MT)
                  (INST-p j) (INST-in j MT)
                  (not (committed-p i)))
             (not (retire-stg-p (INST-stg (step-INST j MT MA sigs)))))
   :hints (("goal" :use ((:instance INST-is-at-one-of-the-stages)
                         (:instance INST-is-at-one-of-the-stages (i j)))
                  :in-theory (disable INST-is-at-one-of-the-stages)))))

(local
 (defthm no-commit-inst-p-step-trace-if-car-is-not-retire-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (tail-p sub trace)
                  (INST-listp trace) (MA-input-p sigs)
                  (INST-listp sub)
                  (not (equal sub trace))
                  (not (b1p (ex-intr? MA sigs)))
                  (not (committed-p (car trace))))
             (no-commit-inst-p
              (step-trace sub MT MA sigs isa spc smc)))
   :hints (("goal" :induct (step-trace sub MT MA sigs isa spc smc)
                  :restrict
                  ((not-commit-stg-p-step-inst-if-earlier-is-committed-p
                    ((i (car trace))))
                   (not-retire-stg-p-step-inst-if-earlier-is-committed-p
                    ((i (car trace))))))
           (when-found (inst-listp trace)
                       (:cases ((consp trace)))))))

(defthm no-commit-inst-p-step-trace-if-car-is-not-retire
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (consp trace)
                  (INST-listp trace) (MA-input-p sigs)
                  (not (b1p (ex-intr? MA sigs)))
                  (not (committed-p (car trace))))
             (no-commit-inst-p
              (step-trace (cdr trace) MT MA sigs isa spc smc))))
)

(defthm trace-all-specultv-p-step-trace
    (implies (and (inv MT MA)
                  (trace-all-specultv-p trace)
                  (MA-state-p MA) (MAETT-p MT)
                  (subtrace-p trace MT)
```

```
                (INST-listp trace)
                (not (b1p (ex-intr? MA sigs)))
                (b1p spc))
           (trace-all-specultv-p
            (step-trace trace MT MA sigs ISA spc smc)))
  :hints (("goal" :in-theory (enable lift-b-ops))))

(defthm trace-correct-speculation-p-step-trace
    (implies (and (inv MT MA)
                  (trace-correct-speculation-p trace)
                  (MA-state-p MA) (MAETT-p MT)
                  (MA-input-p sigs)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (not (b1p spc))
                  (not (trace-CMI-p
                        (step-trace trace MT MA sigs ISA spc smc))))
             (trace-correct-speculation-p (step-trace trace MT MA sigs ISA
                                                      spc smc)))
  :hints (("goal" :in-theory (e/d (lift-b-ops committed-p)
                                  (INST-STG-STEP-IFU-INST-IF-DQ-FULL
                                   INST-is-at-one-of-the-stages
                                   dispatch-commit-inst-stages)))
          (when-found-multiple ((IFU-STG-P (INST-STG (CAR TRACE)))
                                (cdr trace))
                               (:cases ((consp (cdr trace)))))
          (when-found (B1P (INST-EXINTR-NOW? (CAR TRACE) MA SIGS))
                      (:cases ((b1p (ex-intr? MA sigs)))))
          (when-found (TRACE-CMI-P
                       (STEP-TRACE (CDR TRACE)
                                   MT MA SIGS (INST-POST-ISA (CAR TRACE))
                                   (B-IOR (INST-SPECULTV? (CAR TRACE))
                                          (INST-START-SPECULTV? (CAR TRACE)))
                                   (INST-MODIFIED? (CAR TRACE))))
                      (:use (:instance INST-is-at-one-of-the-stages
                                       (i (car trace)))))
          (when-found (TRACE-CORRECT-SPECULATION-P
                       (STEP-TRACE (CDR TRACE)
                                   MT MA SIGS (INST-POST-ISA (CAR TRACE))
                                   (B-IOR (INST-SPECULTV? (CAR TRACE))
                                          (INST-START-SPECULTV? (CAR TRACE)))
                                   (INST-MODIFIED? (CAR TRACE))))
                      (:use (:instance INST-is-at-one-of-the-stages
                                       (i (car trace)))))
          (when-found (B1P (B-IOR (INST-SPECULTV? (CAR TRACE))
                                  (INST-START-SPECULTV? (CAR TRACE))))
                      (:use (:instance INST-is-at-one-of-the-stages
                                       (i (car trace)))))
          (when-found (TRACE-CORRECT-SPECULATION-P (CDR TRACE))
                      (:use (:instance INST-is-at-one-of-the-stages
                                       (i (car trace)))))))

(defthm correct-speculation-p-preserved
    (implies (and (inv MT MA)
                  (MA-state-p MA) (MAETT-p MT) (MA-input-p sigs)
                  (not (MT-CMI-p (MT-step MT MA sigs))))
             (correct-speculation-p (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable correct-speculation-p
                                     MT-CMI-p
                                     inv))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
; Proof of no-specultv-commit-p-preserved
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Proof of no-specultv-commit-p for initial states
(defthm no-specultv-commit-p-init-MT
    (implies (and (MA-state-p MA) (b1p (MA-flushed? MA)))
             (no-specultv-commit-p (init-MT MA)))
  :hints (("goal" :in-theory (enable no-specultv-commit-p init-MT))))

;;;; Step case
(encapsulate nil
(local
(defthm no-specultv-commit-p-preserved-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MA-state-p MA) (MAETT-p MT) (MA-input-p sigs)
                  (not (MT-CMI-p (MT-step MT MA sigs))))
             (trace-no-specultv-commit-p (step-trace trace MT MA sigs
                                                     ISA spc smc)))
  :hints (("Goal" :in-theory (enable lift-b-ops
                                     committed-p
                                     NOT-INST-SPECULTV-INST-IN-IF-COMMITTED))
          (when-found (COMMIT-STG-P (INST-STG (STEP-INST (CAR TRACE)
                                                         MT MA SIGS)))
                      (:cases ((committed-p (car trace)))))
          (when-found (RETIRE-STG-P (INST-STG (STEP-INST (CAR TRACE)
                                                         MT MA SIGS)))
                      (:cases ((committed-p (car trace))))))))

(defthm no-specultv-commit-p-preserved
    (implies (and (inv MT MA)
                  (MA-state-p MA) (MAETT-p MT) (MA-input-p sigs)
                  (not (MT-CMI-p (MT-step MT MA sigs))))
             (no-specultv-commit-p (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable MT-step
                                     no-specultv-commit-p))))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Proof of correct-exintr-p-preserved
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Proof of correct-exintr-p for initial states
(defthm correct-exintr-p-init-MT
    (correct-exintr-p (init-MT MA))
  :hints (("goal" :in-theory (enable init-MT correct-exintr-p))))

;; Invariant proof
(encapsulate nil
(local
(defthm INST-exintr-if-not-retire-help
    (implies (and (member-equal i trace)
                  (trace-correct-exintr-p trace)
                  (INST-p i)
                  (not (retire-stg-p (INST-stg i))))
             (equal (INST-exintr? i) 0))
  :hints (("Goal" :in-theory (enable lift-b-ops equal-b1p-converter)))))

(defthm INST-exintr-if-not-retire
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (retire-stg-p (INST-stg i))))
             (equal (INST-exintr? i) 0))
```

```
  :hints (("Goal" :in-theory (enable inv CORRECT-EXINTR-P INST-in))))
)

(encapsulate nil
(local
(defthm correct-exintr-p-preserved-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA))
             (trace-correct-exintr-p (step-trace trace MT MA sigs
                                                 ISA spc smc)))
  :hints ((when-found (RETIRE-STG-P (INST-STG (STEP-INST (CAR TRACE)
                                                         MT MA SIGS)))
                      (:cases ((retire-stg-p (INST-stg (car trace))))))))))

(defthm correct-exintr-p-preserved
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA))
             (correct-exintr-p (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable correct-exintr-p))))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; misc-inv
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; proof of misc-inv for initial states
(defthm misc-inv-init-MT
    (implies (and (MA-state-p MA) (b1p (MA-flushed? MA)))
             (misc-inv (init-MT MA) MA))
  :hints (("goal" :in-theory (enable init-MT misc-inv lift-b-ops
                                     ROB-empty? MA-flushed?
                                     CORRECT-ENTRIES-IN-DQ-P
                                     DQ-empty?
                                     equal-b1p-converter))))

;; Invariant Proof
(defthm rob-head-MA-rob-MA-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (ROB-head (MA-ROB (MA-step MA sigs)))
                    (MT-ROB-head (MT-step MT MA sigs))))
  :hints (("Goal" :in-theory (enable MT-step step-mt-rob-head))))

(defthm rob-tail-MA-rob-MA-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (ROB-tail (MA-ROB (MA-step MA sigs)))
                    (MT-ROB-tail (MT-step MT MA sigs))))
  :hints (("Goal" :in-theory (enable MT-step step-mt-rob-tail))))

; this should follow mt-dq-len-lt-4
(defthm MT-dq-len-le-4
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA))
             (<= (MT-dq-len MT) 4))
  :hints (("Goal" :in-theory (enable inv misc-inv)))
  :rule-classes :linear)

(defthm MT-DQ-len-le-4-MT-step
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA))
             (<= (MT-DQ-len (MT-step MT MA sigs)) 4))
  :hints (("Goal" :in-theory (enable MT-step STEP-MT-DQ-LEN
                                     lift-b-ops DQ-FULL?)))
```

```
  :rule-classes :linear)

(defthm not-dispatch-inst-if-MT-dq-len-0
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA)
                  (equal (MT-DQ-len MT) 0))
             (equal (dispatch-inst? MA) 0))
  :hints (("goal" :in-theory (enable dispatch-inst? lift-b-ops
                                     dispatch-no-unit?
                                     dispatch-to-IU? dispatch-to-MU?
                                     dispatch-to-BU? dispatch-to-LSU?
                                     DQ-ready-no-unit? DQ-ready-to-IU?
                                     DQ-ready-to-MU? DQ-ready-to-BU?
                                     DQ-ready-to-LSU? equal-b1p-converter))))

(defthm correct-entries-in-DQ-p-preserved
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA))
             (correct-entries-in-DQ-p (MT-step MT MA sigs) (MA-step MA sigs)))
  :hints (("Goal" :in-theory (e/d (correct-entries-in-DQ-p MT-step
                                     step-de0 step-de1 step-de2 step-de3
                                     STEP-MT-DQ-LEN
                                     decode-output DQ-FULL?
                                     de1-out de2-out de3-out
                                     lift-b-ops)
                           ()))))

(defthm rob-flg-MA-step-=-MA-rob-flg-MT-step
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA))
             (equal (ROB-flg (MA-rob (MA-step MA sigs)))
                    (MT-rob-flg (MT-step MT MA sigs))))
  :hints (("Goal" :in-theory (enable step-rob MT-step step-MT-rob-flg))))

(defthm misc-inv-preserved
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA))
             (misc-inv (MT-step MT MA sigs) (MA-step MA sigs)))
  :hints (("Goal" :in-theory (e/d (misc-inv)
                                  (MA-ROB-MA-STEP)))))

(deflabel end-misc-inv)
```

D.6.9 uniq-inv.lisp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MI-inv.lisp
; Author  Jun Sawada, University of Texas at Austin
;
;  This book proves the invariant properties no-stage-conflict and
;  no-tag-conflict.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(in-package "ACL2")

(include-book "MA2-lemmas")
(include-book "MAETT-lemmas")

(deflabel begin-uniq-inv)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Proof of no-stage-conflict
; Prove that no instruction are in the same pipeline latch.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; index
; no-stage-conflict-init-MT
```

```
;   no-IFU-stg-conflict-preserved
;   no-DQ-stg-conflict-preserved
;     uniq-inst-at-DE0-mt-step
;     no-inst-at-DE0-mt-step
;     uniq-inst-at-DE1-mt-step
;     no-inst-at-DE1-mt-step
;     uniq-inst-at-DE2-mt-step
;     no-inst-at-DE2-mt-step
;     uniq-inst-at-DE3-mt-step
;     no-inst-at-DE3-mt-step
;   no-IU-stg-conflict
;     uniq-inst-at-IU-RS0-MT-step
;     no-inst-at-IU-RS0-MT-step
;     uniq-inst-at-IU-RS1-MT-step
;     no-inst-at-IU-RS1-MT-step
;   no-BU-stg-conflict-preserved
;     uniq-inst-at-BU-RS0-MT-step
;     no-inst-at-BU-RS0-MT-step
;     uniq-inst-at-BU-RS1-MT-step
;     no-inst-at-BU-RS1-MT-step
;   no-MU-stg-conflict-preserved
;     uniq-INST-at-MU-RS0-MT-step
;     no-INST-at-MU-RS0-MT-step
;     uniq-INST-at-MU-RS1-MT-step
;     no-INST-at-MU-RS1-MT-step
;     uniq-INST-at-MU-lch1-MT-step
;     no-INST-at-MU-lch1-MT-step
;     uniq-INST-at-MU-lch2-MT-step
;     no-INST-at-MU-lch2-MT-step
;   no-LSU-stg-conflict-preserved
;     uniq-inst-at-LSU-RS0-MT-step
;     no-inst-at-LSU-RS0-MT-step
;     uniq-inst-at-LSU-RS1-MT-step
;     no-inst-at-LSU-RS1-MT-step
;     uniq-inst-at-LSU-rbuf-MT-step
;     no-inst-at-LSU-rbuf-MT-step
;     uniq-inst-at-LSU-wbuf0-MT-step
;     no-inst-at-LSU-wbuf0-MT-step
;     uniq-inst-at-LSU-wbuf1-MT-step
;     no-inst-at-LSU-wbuf1-MT-step
;     uniq-inst-at-LSU-lch-MT-step
;     no-inst-at-LSU-lch-MT-step

;;;; Proof of no-stage-conflict for initial states
(defthm no-stage-conflict-init-MT
    (implies (and (MA-state-p MA) (b1p (MA-flushed? MA)))
             (no-stage-conflict (init-MT MA) MA))
  :Hints (("goal" :in-theory (enable init-MT no-stage-conflict
                                     NO-IFU-STG-CONFLICT
                                     NO-DQ-STG-CONFLICT
                                     NO-IU-STG-CONFLICT
                                     NO-MU-STG-CONFLICT
                                     NO-BU-STG-CONFLICT
                                     NO-LSU-STG-CONFLICT
                                     NO-INST-AT-STGS
                                     MA-flushed? lift-b-ops
                                     IFU-empty? DQ-empty?
                                     IU-empty? BU-empty? LSU-empty?
                                     MU-empty? NO-INST-AT-STG))))

;; Proof of no-inst-at-non-retire-MT-step-if-flush-all
;; This lemma says that after flush-all? is asserted, only committed
```

```
;; instructions can be in a MAETT.

(encapsulate nil
(local
(defthm no-inst-at-non-retire-MT-step-if-flush-all-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (MT-all-commit-before-trace trace MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (flush-all? MA sigs))
                  (not (retire-stg-p stg))
                  (not (commit-stg-p stg)))
             (no-inst-at-stg-in-trace stg (step-trace trace MT MA sigs
                                                      ISA spc smc)))
  :hints ((when-found (INST-CAUSE-JMP? (CAR TRACE) MT MA SIGS)
                      (:cases ((committed-p (car trace)))))
          ("goal" :in-theory (enable committed-p)))))

(defthm no-inst-at-non-retire-MT-step-if-flush-all
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (flush-all? MA sigs))
                  (not (retire-stg-p stg))
                  (not (commit-stg-p stg)))
             (no-inst-at-stg stg (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg MT-step))))
)

;;; Proof of no-IFU-stg-conflict-preserved
;;; There exists an instruction at IFU stage iff IFU-valid? is on.
(encapsulate nil
(local
(defthm uniq-inst-at-IFU-MT-step-help-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (b1p (DQ-full? (MA-dq MA)))
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (IFU-valid? (MA-IFU MA)))
                  (no-inst-at-stg-in-trace '(IFU) trace))
             (no-inst-at-stg-in-trace '(IFU) (step-trace trace MT MA sigs
                                                         ISA spc smc)))
  :hints (("goal" :in-theory (enable IFU-stg-p)))))

(local
(defthm uniq-inst-at-IFU-MT-step-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (b1p (DQ-full? (MA-dq MA)))
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (IFU-valid? (MA-IFU MA)))
                  (uniq-inst-at-stg-in-trace '(IFU) trace))
             (uniq-inst-at-stg-in-trace '(IFU) (step-trace trace MT MA sigs
                                                           ISA spc smc)))
  :hints (("goal" :in-theory (enable IFU-stg-p)))))

(defthm uniq-inst-at-IFU-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
```

```
                (b1p (DQ-full? (MA-dq MA)))
                (not (b1p (flush-all? MA sigs)))
                (b1p (IFU-valid? (MA-IFU MA))))
           (uniq-inst-at-stg '(IFU) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (e/d (uniq-inst-at-stg)
                                  (UNIQ-INST-AT-IFU-IF-IFU-VALID))
                :use ((:instance UNIQ-INST-AT-IFU-IF-IFU-VALID)))))
)

(defthm not-inst-step-inst-if-fetch-inst
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (b1p (fetch-inst? MA sigs))
                (MAETT-p MT) (MA-state-p MA))
           (not (equal (INST-stg (step-inst i MT MA sigs)) '(IFU))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                :in-theory (e/d (fetch-inst? lift-b-ops
                                   new-dq-stage)
                                (inst-is-at-one-of-the-stages)))))


(encapsulate nil
(local
(defthm uniq-inst-at-IFU-MT-step-if-fetch-inst-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (fetch-inst? MA sigs)))
           (uniq-inst-at-stg-in-trace '(IFU)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(defthm uniq-inst-at-IFU-MT-step-if-fetch-inst
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (fetch-inst? MA sigs)))
           (uniq-inst-at-stg '(IFU) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable uniq-inst-at-stg))))
)

(defthm not-INST-stg-step-INST-IFU-if-not-DQ-full
    (implies (and (INST-p i) (not (b1p (DQ-full? (MA-DQ MA)))))
           (not (equal (INST-stg (step-INST i MT MA sigs)) '(IFU))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                :in-theory (e/d (new-dq-stage)
                                (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-IFU-MT-step-if-not-fetch-inst-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (fetch-inst? MA sigs)))
                (not (b1p (DQ-full? (MA-DQ MA)))))
           (no-inst-at-stg-in-trace '(IFU)
                                    (step-trace trace MT MA sigs
                                                ISA spc smc)))))

(defthm no-inst-at-IFU-MT-step-if-not-fetch-inst
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
```

```
                (not (b1p (fetch-inst? MA sigs)))
                (not (b1p (DQ-full? (MA-DQ MA)))))
           (no-inst-at-stg '(IFU) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg))))
)

(encapsulate nil
(local
(defthm no-inst-at-IFU-MT-step-help
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (fetch-inst? MA sigs)))
                (no-inst-at-stg-in-trace '(IFU) trace))
           (no-inst-at-stg-in-trace '(IFU) (step-trace trace MT MA sigs
                                                       ISA spc smc)))
  :hints (("goal" :in-theory (enable IFU-stg-p)))))

(defthm no-inst-at-IFU-MT-step
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (fetch-inst? MA sigs)))
                (no-inst-at-stg '(IFU) MT))
           (no-inst-at-stg '(IFU) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable MT-step no-inst-at-stg))))
)

(defthm no-IFU-stg-conflict-preserved
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA))
           (no-IFU-stg-conflict (MT-step MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable no-IFU-stg-conflict fetch-inst?
                                     step-IFU lift-b-ops))))

;;;  Proof of no-DQ-stg-conflict-preserved
;; Proof of uniq-inst-at-DE0-mt-step
(defthm not-INST-stg-step-inst-DE0-if-not-DE0-valid
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (not (IFU-stg-p (INST-stg i)))
                (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))
                (MAETT-p MT) (MA-state-p MA))
           (not (equal (INST-stg (step-inst i MT MA sigs)) '(DQ 0))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                :in-theory (disable inst-is-at-one-of-the-stages))))

(encapsulate nil
(local
(defthm uniq-inst-at-DE0-if-IFU-valid-help-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT)
                (INST-listp trace)
                (b1p (IFU-valid? (MA-IFU MA)))
                (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))
                (not (b1p (flush-all? MA sigs)))
                (no-inst-at-stg-in-trace '(IFU) trace)
                (MAETT-p MT) (MA-state-p MA))
           (no-inst-at-stg-in-trace '(DQ 0) (step-trace trace MT MA sigs
                                                        ISA spc smc)))
  :hints (("goal" :in-theory (enable lift-b-ops dq-full? new-dq-stage
                                     IFU-stg-p)))))

(local
(defthm uniq-inst-at-DE0-if-IFU-valid-help
```

```
    (implies (and (inv MT MA)
                  (subtrace-p trace MT)
                  (b1p (IFU-valid? (MA-IFU MA)))
                  (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))
                  (not (b1p (flush-all? MA sigs)))
                  (INST-listp trace)
                  (uniq-inst-at-stg-in-trace '(IFU) trace)
                  (MAETT-p MT) (MA-state-p MA))
             (uniq-inst-at-stg-in-trace '(DQ 0) (step-trace trace MT MA sigs
                                                            ISA spc smc)))
  :hints (("goal" :in-theory (enable lift-b-ops dq-full? new-dq-stage
                                     IFU-stg-p)))))

(defthm uniq-inst-at-DE0-MT-step-if-IFU-valid
    (implies (and (inv MT MA)
                  (b1p (IFU-valid? (MA-IFU MA)))
                  (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))
                  (not (b1p (flush-all? MA sigs)))
                  (MAETT-p MT) (MA-state-p MA))
             (uniq-inst-at-stg '(DQ 0) (MT-step MT MA sigs)))
  :hints (("goal" :use ((:instance UNIQ-INST-AT-IFU-IF-IFU-VALID))
                  :in-theory (e/d (uniq-inst-at-stg)
                                  (UNIQ-INST-AT-IFU-IF-IFU-VALID)))))
)

(defthm not-INST-stg-step-inst-DE0-if-DE1-empty
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (IFU-stg-p (INST-stg i)))
                  (not (b1p (DE-valid? (DQ-DE1 (MA-DQ MA)))))
                  (b1p (dispatch-inst? MA))
                  (MAETT-p MT) (MA-state-p MA))
             (not (equal (INST-stg (step-inst i MT MA sigs)) '(DQ 0))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (dq-stg-p)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-DE0-if-DE1-empty-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (b1p (IFU-valid? (MA-IFU MA)))
                  (not (b1p (DE-valid? (DQ-DE1 (MA-DQ MA)))))
                  (b1p (dispatch-inst? MA))
                  (no-inst-at-stg-in-trace '(IFU) trace)
                  (not (b1p (flush-all? MA sigs)))
                  (MAETT-p MT) (MA-state-p MA))
             (no-inst-at-stg-in-trace '(DQ 0)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))
  :hints (("goal" :in-theory (enable lift-b-ops dq-full? new-dq-stage
                                     IFU-stg-p)))))

(local
(defthm uniq-inst-at-DE0-if-DE1-empty-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (b1p (IFU-valid? (MA-IFU MA)))
                  (not (b1p (DE-valid? (DQ-DE1 (MA-DQ MA)))))
                  (b1p (dispatch-inst? MA))
                  (uniq-inst-at-stg-in-trace '(IFU) trace)
```

```
                (not (b1p (flush-all? MA sigs)))
                (MAETT-p MT) (MA-state-p MA))
           (uniq-inst-at-stg-in-trace '(DQ 0)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))
  :hints (("goal" :in-theory (enable lift-b-ops dq-full? new-dq-stage
                                     IFU-stg-p))
          (when-found-multiple ((B1P (DE-VALID? (DQ-DE1 (MA-DQ MA))))
                                (B1P (DISPATCH-INST? MA))
                                (MT-DQ-LEN MT))
                       (:cases ((b1p (DE-VALID? (DQ-DE0 (MA-DQ MA))))))))))

(defthm uniq-inst-at-DE0-MT-step-if-DE1-empty
    (implies (and (inv MT MA)
                  (b1p (IFU-valid? (MA-IFU MA)))
                  (not (b1p (DE-valid? (DQ-DE1 (MA-DQ MA)))))
                  (b1p (dispatch-inst? MA))
                  (not (b1p (flush-all? MA sigs)))
                  (MAETT-p MT) (MA-state-p MA))
             (uniq-inst-at-stg '(DQ 0) (MT-step MT MA sigs)))
    :hints (("goal" :use ((:instance UNIQ-INST-AT-IFU-IF-IFU-VALID))
                  :in-theory (e/d (uniq-inst-at-stg)
                                  (UNIQ-INST-AT-IFU-IF-IFU-VALID)))))
)

(defthm not-INST-stg-step-inst-DE0-if-DE1-valid
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (equal (INST-stg i) '(DQ 1)))
                  (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))
                  (b1p (dispatch-inst? MA))
                  (MAETT-p MT) (MA-state-p MA))
             (not (equal (INST-stg (step-inst i MT MA sigs)) '(DQ 0))))
  :hints (("goal" :use ((:instance inst-is-at-one-of-the-stages)
                        (:instance mt-dq-len-ge-2))
                  :in-theory (e/d (dq-stg-p new-dq-stage COERCE-DQ-STG)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-DE0-if-DE1-valid-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))
                  (b1p (dispatch-inst? MA))
                  (not (b1p (flush-all? MA sigs)))
                  (no-inst-at-stg-in-trace '(DQ 1) trace)
                  (MAETT-p MT) (MA-state-p MA))
             (no-inst-at-stg-in-trace '(DQ 0) (step-trace trace MT MA sigs
                                                          ISA spc smc)))))

(local
(defthm uniq-inst-at-DE0-if-DE1-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))
                  (b1p (dispatch-inst? MA))
                  (not (b1p (flush-all? MA sigs)))
                  (uniq-inst-at-stg-in-trace '(DQ 1) trace)
                  (MAETT-p MT) (MA-state-p MA))
             (uniq-inst-at-stg-in-trace '(DQ 0) (step-trace trace MT MA sigs
                                                            ISA spc smc)))))
```

```
(defthm uniq-inst-at-DE0-MT-step-if-DE1-valid
    (implies (and (inv MT MA)
                  (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))
                  (b1p (dispatch-inst? MA))
                  (not (b1p (flush-all? MA sigs)))
                  (MAETT-p MT) (MA-state-p MA))
             (uniq-inst-at-stg '(DQ 0) (MT-step MT MA sigs)))
      :hints (("goal" :use ((:instance UNIQ-INST-AT-STG-IF-DQ-DE1-valid))
                  :in-theory (e/d (uniq-inst-at-stg)
                                  (UNIQ-INST-AT-STG-IF-DQ-DE1-valid)))))
)

(defthm not-INST-stg-step-inst-DE0-if-DE0-valid
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (equal (INST-stg i) '(DQ 0)))
                  (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))
                  (not (b1p (dispatch-inst? MA)))
                  (MAETT-p MT) (MA-state-p MA))
             (not (equal (INST-stg (step-inst i MT MA sigs)) '(DQ 0))))
  :hints (("goal" :use ((:instance inst-is-at-one-of-the-stages)
                        (:instance MT-dq-len-ge-1))
                  :in-theory (e/d (dq-stg-p new-dq-stage coerce-dq-stg)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-DE0-MT-step-if-dq-de0-valid-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))
                  (not (b1p (dispatch-inst? MA)))
                  (not (b1p (flush-all? MA sigs)))
                  (no-inst-at-stg-in-trace '(DQ 0) trace)
                  (MAETT-p MT) (MA-state-p MA))
             (no-inst-at-stg-in-trace '(DQ 0) (step-trace trace MT MA sigs
                                                          ISA spc smc)))))
(local
(defthm uniq-inst-at-DE0-MT-step-if-dq-de0-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))
                  (not (b1p (dispatch-inst? MA)))
                  (not (b1p (flush-all? MA sigs)))
                  (uniq-inst-at-stg-in-trace '(DQ 0) trace)
                  (MAETT-p MT) (MA-state-p MA))
             (uniq-inst-at-stg-in-trace '(DQ 0) (step-trace trace MT MA sigs
                                                            ISA spc smc)))))

(defthm uniq-inst-at-DE0-MT-step-if-dq-de0-valid
    (implies (and (inv MT MA)
                  (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))
                  (not (b1p (dispatch-inst? MA)))
                  (not (b1p (flush-all? MA sigs)))
                  (MAETT-p MT) (MA-state-p MA))
             (uniq-inst-at-stg '(DQ 0) (MT-step MT MA sigs)))
  :hints (("goal" :use ((:instance UNIQ-INST-AT-STG-IF-DQ-DE0-valid))
                  :in-theory (e/d (uniq-inst-at-stg)
                                  (UNIQ-INST-AT-STG-IF-DQ-DE0-valid)))))
)
```

```
(defthm uniq-inst-at-DE0-mt-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (DE-valid? (DQ-DE0 (MA-DQ (MA-step MA sigs))))))
             (uniq-INST-at-stg '(DQ 0) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-dq step-DE0 lift-b-ops
                                     decode-output
                                     DE3-out DE2-out DE1-out))))

;; Proof of no-inst-at-DE0-mt-step
(encapsulate nil
(local
(defthm no-inst-at-DE0-mt-step-if-not-IFU-valid-help-lemma1
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(IFU) trace)
                  (no-inst-at-stg-in-trace '(DQ 0) trace)
                  (no-inst-at-stg-in-trace '(DQ 1) trace)
                  (no-inst-at-stg-in-trace '(DQ 2) trace)
                  (no-inst-at-stg-in-trace '(DQ 3) trace))
             (no-inst-at-stg-in-trace '(DQ 0)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))
  :hints (("goal" :in-theory (enable IFU-stg-p DQ-stg-p)))))

(defthm no-inst-at-DE0-mt-step-if-not-IFU-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))
                  (not (b1p (IFU-valid? (MA-IFU MA)))))
             (no-inst-at-stg '(DQ 0) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg)
                  :use ((:instance NO-INST-AT-IFU-IF-IFU-INVALID)
                        (:instance NO-INST-AT-STG-IF-NO-DQ-DE0-VALID)
                        (:instance NO-INST-AT-STG-IF-NO-DQ-DE1-VALID)
                        (:instance NO-INST-AT-STG-IF-NO-DQ-DE2-VALID)
                        (:instance NO-INST-AT-STG-IF-NO-DQ-DE3-VALID)))))
)

(defthm not-INST-stg-step-INST-DE0-if-not-IFU-DE1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (not (equal (INST-stg i) '(IFU)))
                  (not (equal (INST-stg i) '(DQ 1))))
             (not (equal (inst-stg (step-INST i MT MA sigs)) '(DQ 0))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (dq-stg-p new-dq-stage IFU-stg-p
                                            DQ-stg-p coerce-dq-stg)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-DE0-mt-step-if-dispatch-inst-help-lemma1
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (no-inst-at-stg-in-trace '(IFU) trace)
                  (no-inst-at-stg-in-trace '(DQ 1) trace))
```

```
            (no-inst-at-stg-in-trace '(DQ 0)
                                     (step-trace trace MT MA sigs
                                                 ISA spc smc)))))

(defthm no-inst-at-DE0-mt-step-if-dispatch-inst
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (not (b1p (DE-valid? (DQ-DE1 (MA-DQ MA)))))
                  (not (b1p (IFU-valid? (MA-IFU MA)))))
             (no-inst-at-stg '(DQ 0) (MT-step MT MA sigs)))
  :hints (("goal" :use ((:instance NO-INST-AT-IFU-IF-IFU-INVALID)
                        (:instance NO-INST-AT-STG-IF-NO-DQ-DE1-VALID))
                  :in-theory (enable no-inst-at-stg))))
)

(defthm no-inst-at-DE0-mt-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (DE-valid? (DQ-DE0 (MA-DQ (MA-step MA sigs)))))))
             (no-INST-at-stg '(DQ 0) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-dq step-DE0 lift-b-ops
                                     decode-output
                                     DE3-out DE2-out DE1-out))))

;; Proof of uniq-inst-at-DE1-mt-step
(defthm not-INST-stg-step-INST-DE1-if-not-dispatch-inst
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (dispatch-inst? MA)))
                  (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))
                  (not (equal (INST-stg i) '(DQ 1))))
             (not (equal (inst-stg (step-INST i MT MA sigs)) '(DQ 1))))
  :hints (("goal" :use ((:instance inst-is-at-one-of-the-stages)
                        (:instance mt-dq-len-ge-2))
                  :in-theory (e/d (dq-stg-p new-dq-stage IFU-stg-p
                                    DQ-stg-p coerce-dq-stg)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-DE1-MT-step-if-DE1-valid-help-lemma2
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (dispatch-inst? MA)))
                  (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))
                  (no-inst-at-stg-in-trace '(DQ 1) trace))
             (no-inst-at-stg-in-trace '(DQ 1)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
(defthm uniq-inst-at-DE1-MT-step-if-DE1-valid-help-lemma1
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (dispatch-inst? MA)))
                  (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))
                  (not (b1p (flush-all? MA sigs)))
                  (uniq-inst-at-stg-in-trace '(DQ 1) trace))
```

```
            (uniq-inst-at-stg-in-trace '(DQ 1)
                                       (step-trace trace MT MA sigs
                                                   ISA spc smc)))))

(defthm uniq-inst-at-DE1-MT-step-if-DE1-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (dispatch-inst? MA)))
                  (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stg '(DQ 1) (MT-step MT MA sigs)))
  :hints (("goal" :use ((:instance UNIQ-INST-AT-STG-IF-DQ-DE1-VALID))
                  :in-theory (e/d (uniq-inst-at-stg)
                                  (UNIQ-INST-AT-STG-IF-DQ-DE1-VALID)))))
)

(defthm not-INST-stg-step-INST-DE2-if-DE2-valid
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))
                  (not (equal (INST-stg i) '(DQ 2))))
             (not (equal (inst-stg (step-INST i MT MA sigs)) '(DQ 1))))
  :hints (("goal" :use ((:instance inst-is-at-one-of-the-stages)
                        (:instance mt-dq-len-ge-3))
                  :in-theory (e/d (dq-stg-p new-dq-stage IFU-stg-p
                                    DQ-stg-p coerce-dq-stg)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-DE1-MT-step-if-de2-valid-help-lemma2
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))
                  (no-inst-at-stg-in-trace '(DQ 2) trace))
             (no-inst-at-stg-in-trace '(DQ 1)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
(defthm uniq-inst-at-DE1-MT-step-if-de2-valid-help-lemma1
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))
                  (not (b1p (flush-all? MA sigs)))
                  (uniq-inst-at-stg-in-trace '(DQ 2) trace))
             (uniq-inst-at-stg-in-trace '(DQ 1)
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc)))))

(defthm uniq-inst-at-DE1-MT-step-if-de2-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))
                  (not (b1p (flush-all? MA sigs))))
```

```
            (uniq-inst-at-stg '(DQ 1) (MT-step MT MA sigs)))
  :hints (("goal" :use ((:instance UNIQ-INST-AT-STG-IF-DQ-DE2-VALID))
                  :in-theory (e/d (uniq-inst-at-stg)
                                  (UNIQ-INST-AT-STG-IF-DQ-DE2-VALID)))))
)

(defthm not-INST-stg-step-INST-DE1-if-IFU
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (or (and (not (b1p (dispatch-inst? MA)))
                           (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))
                           (not (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))))
                      (and (b1p (dispatch-inst? MA))
                           (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))
                           (not (b1p (DE-valid? (DQ-DE2 (MA-DQ MA)))))))
                  (not (equal (INST-stg i) '(IFU))))
             (not (equal (inst-stg (step-INST i MT MA sigs)) '(DQ 1))))
  :hints (("goal" :use ((:instance inst-is-at-one-of-the-stages))
                  :in-theory (e/d (dq-stg-p new-dq-stage IFU-stg-p
                                    DQ-stg-p coerce-dq-stg)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-DE1-MT-step-if-DE1-empty-help-lemma2
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (or (and (not (b1p (dispatch-inst? MA)))
                           (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))
                           (not (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))))
                      (and (b1p (dispatch-inst? MA))
                           (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))
                           (not (b1p (DE-valid? (DQ-DE2 (MA-DQ MA)))))))
                  (no-inst-at-stg-in-trace '(IFU) trace))
             (no-inst-at-stg-in-trace '(DQ 1)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
(defthm uniq-inst-at-DE1-MT-step-if-DE1-empty-help-lemma1
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (or (and (not (b1p (dispatch-inst? MA)))
                           (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))
                           (not (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))))
                      (and (b1p (dispatch-inst? MA))
                           (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))
                           (not (b1p (DE-valid? (DQ-DE2 (MA-DQ MA)))))))
                  (not (b1p (flush-all? MA sigs)))
                  (uniq-inst-at-stg-in-trace '(IFU) trace))
             (uniq-inst-at-stg-in-trace '(DQ 1)
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc)))
  :hints (("goal" :in-theory (enable DQ-FULL? new-dq-stage lift-b-ops)))))

(defthm uniq-inst-at-DE1-MT-step-if-DE1-empty
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (or (and (not (b1p (dispatch-inst? MA)))
```

```
                          (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))
                          (not (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))))
                     (and (b1p (dispatch-inst? MA))
                          (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))
                          (not (b1p (DE-valid? (DQ-DE2 (MA-DQ MA)))))))
                (b1p (IFU-valid? (MA-IFU MA)))
                (not (b1p (flush-all? MA sigs))))
           (uniq-inst-at-stg '(DQ 1) (MT-step MT MA sigs)))
    :hints (("goal" :use (
                        (:instance UNIQ-INST-AT-IFU-IF-IFU-VALID))
                :in-theory (e/d (uniq-inst-at-stg)
                                (UNIQ-INST-AT-IFU-IF-IFU-VALID)))))
)

(defthm uniq-inst-at-DE1-MT-step
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (DE-valid? (DQ-DE1 (MA-DQ (MA-step MA sigs))))))
           (uniq-INST-at-stg '(DQ 1) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-dq step-DE1 lift-b-ops
                                     decode-output DE3-out DE2-out))))

;; Proof of no-inst-at-DE1-mt-step
(defthm not-INST-stg-step-INST-DE1-if-not-DQ-stg-p
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (INST-in i MT) (INST-p i)
                (not (DQ-stg-p (INST-stg i)))
                (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))))
           (not (equal (INST-stg (step-INST i MT MA sigs)) '(DQ 1))))
  :hints  (("goal" :use ((:instance inst-is-at-one-of-the-stages)
                         (:instance MT-DQ-len-ge-1))
                  :in-theory (e/d (dq-stg-p new-dq-stage IFU-stg-p
                                    DQ-stg-p coerce-dq-stg)
                                  (inst-is-at-one-of-the-stages)))))


(encapsulate nil
(local
(defthm no-inst-at-DE1-mt-step-if-not-DE0-valid-help
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (subtrace-p trace MT) (INST-listp trace)
                (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA)))))
                (no-INST-at-stg-in-trace '(DQ 0) trace)
                (no-INST-at-stg-in-trace '(DQ 1) trace)
                (no-INST-at-stg-in-trace '(DQ 2) trace)
                (no-INST-at-stg-in-trace '(DQ 3) trace))
           (no-INST-at-stg-in-trace '(DQ 1) (step-trace trace MT MA sigs
                                                        ISA spc smc)))
  :hints (("goal" :in-theory (enable DQ-stg-p)))))

(defthm no-inst-at-DE1-mt-step-if-not-DE0-valid
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (DE-valid? (DQ-DE0 (MA-DQ MA))))))
           (no-INST-at-stg '(DQ 1) (MT-step MT MA sigs)))
  :hints (("goal" :use ((:instance NO-INST-AT-STG-IF-NO-DQ-DE0-VALID)
                        (:instance NO-INST-AT-STG-IF-NO-DQ-DE1-VALID)
                        (:instance NO-INST-AT-STG-IF-NO-DQ-DE2-VALID)
                        (:instance NO-INST-AT-STG-IF-NO-DQ-DE3-VALID))
                :in-theory (e/d (no-inst-at-stg) ()))))
```

```
)

(defthm not-INST-stg-step-INST-DE1-if-not-DE2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (not (equal (INST-stg i) '(DQ 2)))
                  (not (b1p (DE-valid? (DQ-DE1 (MA-DQ MA)))))
                  (b1p (dispatch-inst? MA)))
             (not (equal (INST-stg (step-INST i MT MA sigs)) '(DQ 1))))
  :hints  (("goal" :use ((:instance inst-is-at-one-of-the-stages)
                         (:instance MT-DQ-len-lt-2))
                   :in-theory (e/d (dq-stg-p new-dq-stage IFU-stg-p
                                     DQ-stg-p coerce-dq-stg)
                                   (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-DE1-mt-step-if-not-DE1-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (no-inst-at-stg-in-trace '(DQ 2) trace)
                  (not (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))))
             (no-INST-at-stg-in-trace '(DQ 1) (step-trace trace MT MA sigs
                                                          ISA spc smc)))))

(defthm no-inst-at-DE1-mt-step-if-not-DE1-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (not (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))))
             (no-INST-at-stg '(DQ 1) (MT-step MT MA sigs)))
  :hints (("goal" :use ((:instance NO-INST-AT-STG-IF-NO-DQ-DE2-VALID))
                  :in-theory (e/d (no-inst-at-stg) ()))))
)

(defthm not-INST-stg-step-INST-DE1-if-not-IFU-DE1-DE2
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (not (equal (INST-stg i) '(IFU)))
                  (not (equal (INST-stg i) '(DQ 1)))
                  (not (equal (INST-stg i) '(DQ 2))))
             (not (equal (INST-stg (step-INST i MT MA sigs)) '(DQ 1))))
  :hints  (("goal" :use ((:instance inst-is-at-one-of-the-stages))
                   :in-theory (e/d (dq-stg-p new-dq-stage IFU-stg-p
                                     DQ-stg-p coerce-dq-stg
                                     STEP-INST-DQ-INST step-inst-dq
                                     dispatch-inst)
                                   (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-DE1-mt-step-if-not-IFU-DE1-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(DQ 1) trace)
                  (no-inst-at-stg-in-trace '(DQ 2) trace)
                  (no-inst-at-stg-in-trace '(IFU) trace))
```

```
            (no-INST-at-stg-in-trace '(DQ 1) (step-trace trace MT MA sigs
                                                         ISA spc smc)))))

(defthm no-inst-at-DE1-mt-step-if-not-IFU-DE1-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (DE-valid? (DQ-DE1 (MA-DQ MA)))))
                  (not (b1p (IFU-valid? (MA-IFU MA)))))
             (no-INST-at-stg '(DQ 1) (MT-step MT MA sigs)))
  :hints (("goal" :use ((:instance NO-INST-AT-STG-IF-NO-DQ-DE1-VALID)
                        (:instance NO-INST-AT-STG-IF-NO-DQ-DE2-VALID)
                        (:instance NO-INST-AT-IFU-IF-IFU-INVALID))
                  :in-theory (enable no-inst-at-stg))))
)

(defthm not-INST-stg-step-inst-DE1-if-not-IFU-DE2
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (b1p (dispatch-inst? MA))
                  (not (equal (INST-stg i) '(IFU)))
                  (not (equal (INST-stg i) '(DQ 2))))
             (not (equal (INST-stg (step-INST i MT MA sigs)) '(DQ 1))))
  :hints (("goal" :use ((:instance inst-is-at-one-of-the-stages))
                  :in-theory (e/d (new-dq-stage coerce-dq-stg
                                     IFU-stg-p DQ-stg-p)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-DE1-mt-step-if-not-IFU-DE2-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (no-INST-at-stg-in-trace '(DQ 2) trace)
                  (no-INST-at-stg-in-trace '(IFU) trace))
             (no-INST-at-stg-in-trace '(DQ 1)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(defthm no-inst-at-DE1-mt-step-if-not-IFU-DE2-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (not (b1p (DE-valid? (DQ-DE2 (MA-DQ MA)))))
                  (not (b1p (IFU-valid? (MA-IFU MA)))))
             (no-INST-at-stg '(DQ 1) (MT-step MT MA sigs)))
    :hints (("goal" :in-theory (enable no-INST-at-stg)
                    :use ((:instance NO-INST-AT-STG-IF-NO-DQ-DE2-VALID)
                          (:instance NO-INST-AT-IFU-IF-IFU-INVALID)))))
)

(defthm no-inst-at-DE1-mt-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (DE-valid? (DQ-DE1 (MA-DQ (MA-step MA sigs)))))))
             (no-INST-at-stg '(DQ 1) (MT-step MT MA sigs)))
    :hints (("goal" :in-theory (enable step-dq step-DE1 lift-b-ops
                                       decode-output DE3-out DE2-out))))

;; Proof of uniq-inst-at-DE2-mt-step
(defthm not-INST-stg-step-INST-DE2-if-not-dispatch-inst
```

```
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (dispatch-inst? MA)))
                  (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))
                  (not (equal (INST-stg i) '(DQ 2))))
             (not (equal (inst-stg (step-INST i MT MA sigs)) '(DQ 2))))
  :hints (("goal" :use ((:instance inst-is-at-one-of-the-stages)
                        (:instance mt-dq-len-ge-3))
                  :in-theory (e/d (dq-stg-p new-dq-stage IFU-stg-p
                                    DQ-stg-p coerce-dq-stg)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-DE1-MT-step-if-DE1-valid-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (dispatch-inst? MA)))
                  (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))
                  (no-inst-at-stg-in-trace '(DQ 2) trace))
             (no-inst-at-stg-in-trace '(DQ 2)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
(defthm uniq-inst-at-DE2-MT-step-if-de2-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (dispatch-inst? MA)))
                  (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))
                  (not (b1p (flush-all? MA sigs)))
                  (uniq-inst-at-stg-in-trace '(DQ 2) trace))
             (uniq-inst-at-stg-in-trace '(DQ 2)
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc)))))

(defthm uniq-inst-at-DE2-MT-step-if-de2-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (dispatch-inst? MA)))
                  (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stg '(DQ 2) (MT-step MT MA sigs)))
  :hints (("goal" :use ((:instance UNIQ-INST-AT-STG-IF-DQ-DE2-VALID))
                  :in-theory (e/d (uniq-inst-at-stg)
                                  (UNIQ-INST-AT-STG-IF-DQ-DE2-VALID)))))
)

(defthm not-INST-stg-step-INST-DE2-if-DE3-valid
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (b1p (DE-valid? (DQ-DE3 (MA-DQ MA))))
                  (not (equal (INST-stg i) '(DQ 3))))
             (not (equal (inst-stg (step-INST i MT MA sigs)) '(DQ 2))))
  :hints (("goal" :use ((:instance inst-is-at-one-of-the-stages))
                  :in-theory (e/d (dq-stg-p new-dq-stage IFU-stg-p
                                    DQ-stg-p coerce-dq-stg)
```

```
                                (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-DE1-MT-step-if-de2-valid-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (b1p (DE-valid? (DQ-DE3 (MA-DQ MA))))
                  (no-inst-at-stg-in-trace '(DQ 3) trace))
             (no-inst-at-stg-in-trace '(DQ 2)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
(defthm uniq-inst-at-DE2-MT-step-if-de3-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (b1p (DE-valid? (DQ-DE3 (MA-DQ MA))))
                  (not (b1p (flush-all? MA sigs)))
                  (uniq-inst-at-stg-in-trace '(DQ 3) trace))
             (uniq-inst-at-stg-in-trace '(DQ 2)
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc)))))

(defthm uniq-inst-at-DE1-MT-step-if-de3-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (b1p (DE-valid? (DQ-DE3 (MA-DQ MA))))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stg '(DQ 2) (MT-step MT MA sigs)))
  :hints (("goal" :use ((:instance UNIQ-INST-AT-STG-IF-DQ-DE3-VALID))
                  :in-theory (e/d (uniq-inst-at-stg)
                                  (UNIQ-INST-AT-STG-IF-DQ-DE3-VALID)))))
)

(defthm not-INST-stg-step-INST-DE2-if-IFU
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (or (and (not (b1p (dispatch-inst? MA)))
                           (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))
                           (not (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))))
                      (and (b1p (dispatch-inst? MA))
                           (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))
                           (not (b1p (DE-valid? (DQ-DE3 (MA-DQ MA))))))))
                  (not (equal (INST-stg i) '(IFU))))
             (not (equal (inst-stg (step-INST i MT MA sigs)) '(DQ 2))))
  :hints (("goal" :use ((:instance inst-is-at-one-of-the-stages))
                  :in-theory (e/d (dq-stg-p new-dq-stage IFU-stg-p
                                    DQ-stg-p coerce-dq-stg)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-DE1-MT-step-if-DE1-empty-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
```

```
                (MAETT-p MT) (MA-state-p MA)
                (or (and (not (b1p (dispatch-inst? MA)))
                         (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))
                         (not (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))))
                    (and (b1p (dispatch-inst? MA))
                         (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))
                         (not (b1p (DE-valid? (DQ-DE3 (MA-DQ MA)))))))
                (no-inst-at-stg-in-trace '(IFU) trace))
           (no-inst-at-stg-in-trace '(DQ 2)
                                    (step-trace trace MT MA sigs
                                                ISA spc smc)))))

(local
(defthm uniq-inst-at-DE2-MT-step-if-DE1-empty-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (or (and (not (b1p (dispatch-inst? MA)))
                         (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))
                         (not (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))))
                    (and (b1p (dispatch-inst? MA))
                         (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))
                         (not (b1p (DE-valid? (DQ-DE3 (MA-DQ MA)))))))
                (not (b1p (flush-all? MA sigs)))
                (uniq-inst-at-stg-in-trace '(IFU) trace))
           (uniq-inst-at-stg-in-trace '(DQ 2)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))
  :hints (("goal" :in-theory (enable DQ-FULL? new-dq-stage lift-b-ops)))))

(defthm uniq-inst-at-DE1-MT-step-if-de2-empty
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (or (and (not (b1p (dispatch-inst? MA)))
                         (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))
                         (not (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))))
                    (and (b1p (dispatch-inst? MA))
                         (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))
                         (not (b1p (DE-valid? (DQ-DE3 (MA-DQ MA)))))))
                (b1p (IFU-valid? (MA-IFU MA)))
                (not (b1p (flush-all? MA sigs))))
           (uniq-inst-at-stg '(DQ 2) (MT-step MT MA sigs)))
    :hints (("goal" :use ((:instance UNIQ-INST-AT-IFU-IF-IFU-VALID))
                :in-theory (e/d (uniq-inst-at-stg)
                                (UNIQ-INST-AT-IFU-IF-IFU-VALID)))))
)

(defthm uniq-inst-at-DE2-mt-step
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (DE-valid? (DQ-DE2 (MA-DQ (MA-step MA sigs))))))
           (uniq-INST-at-stg '(DQ 2) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-dq step-DE2 lift-b-ops
                                     decode-output DE3-out))))

;; Proof of no-inst-at-DE2-mt-step
(defthm not-INST-stg-step-INST-DE2-if-not-DQ-stg-p
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (INST-in i MT) (INST-p i)
                (not (equal (INST-stg i) '(DQ 2)))
                (not (equal (INST-stg i) '(DQ 3)))
```

```
                (not (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))))
            (not (equal (INST-stg (step-INST i MT MA sigs)) '(DQ 2))))
  :hints  (("goal" :use ((:instance inst-is-at-one-of-the-stages)
                         (:instance MT-DQ-len-lt-2))
                  :in-theory (e/d (dq-stg-p new-dq-stage IFU-stg-p
                                            step-INST-DQ-INST
                                            step-inst-dq dispatch-inst
                                            DQ-stg-p coerce-dq-stg)
                                  (inst-is-at-one-of-the-stages)))))


(encapsulate nil
(local
(defthm no-inst-at-DE2-mt-step-if-not-DE1-valid-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (not (b1p (DE-valid? (DQ-DE1 (MA-DQ MA)))))
                  (no-INST-at-stg-in-trace '(DQ 2) trace)
                  (no-INST-at-stg-in-trace '(DQ 3) trace))
             (no-INST-at-stg-in-trace '(DQ 2) (step-trace trace MT MA sigs
                                                          ISA spc smc)))
  :hints (("goal" :in-theory (enable DQ-stg-p)))))

(defthm no-inst-at-DE2-mt-step-if-not-DE1-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (DE-valid? (DQ-DE1 (MA-DQ MA))))))
             (no-INST-at-stg '(DQ 2) (MT-step MT MA sigs)))
  :hints (("goal" :use ((:instance NO-INST-AT-STG-IF-NO-DQ-DE2-VALID)
                        (:instance NO-INST-AT-STG-IF-NO-DQ-DE3-VALID))
                  :in-theory (e/d (no-inst-at-stg) ()))))
)

(defthm not-INST-stg-step-INST-DE2-if-not-DE3
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (INST-in i MT) (INST-p i)
                  (not (equal (INST-stg i) '(DQ 3)))
                  (not (b1p (DE-valid? (DQ-DE2 (MA-DQ MA)))))
                  (b1p (dispatch-inst? MA)))
             (not (equal (INST-stg (step-INST i MT MA sigs)) '(DQ 2))))
  :hints  (("goal" :use ((:instance inst-is-at-one-of-the-stages)
                         (:instance MT-DQ-len-lt-3))
                  :in-theory (e/d (dq-stg-p new-dq-stage IFU-stg-p
                                            DQ-stg-p coerce-dq-stg)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-DE2-mt-step-if-not-DE2-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (no-inst-at-stg-in-trace '(DQ 3) trace)
                  (not (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))))
             (no-INST-at-stg-in-trace '(DQ 2) (step-trace trace MT MA sigs
                                                          ISA spc smc)))))

(defthm no-inst-at-DE2-mt-step-if-not-DE2-valid
    (implies (and (inv MT MA)
```

```
                (MAETT-p MT) (MA-state-p MA)
                (b1p (dispatch-inst? MA))
                (not (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))))
           (no-INST-at-stg '(DQ 2) (MT-step MT MA sigs)))
  :hints (("goal" :use ((:instance NO-INST-AT-STG-IF-NO-DQ-DE3-VALID))
                :in-theory (e/d (no-inst-at-stg) ()))))

(defthm not-INST-stg-step-INST-DE2-if-not-IFU-DE1-DE3
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (INST-in i MT) (INST-p i)
                (not (equal (INST-stg i) '(IFU)))
                (not (equal (INST-stg i) '(DQ 2)))
                (not (equal (INST-stg i) '(DQ 3))))
           (not (equal (INST-stg (step-INST i MT MA sigs)) '(DQ 2))))
  :hints  (("goal" :use ((:instance inst-is-at-one-of-the-stages))
                 :in-theory (e/d (dq-stg-p new-dq-stage IFU-stg-p
                                   DQ-stg-p coerce-dq-stg
                                   STEP-INST-DQ-INST step-inst-dq
                                   dispatch-inst)
                                 (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-DE2-mt-step-if-not-IFU-DE2-valid-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (no-inst-at-stg-in-trace '(DQ 2) trace)
                (no-inst-at-stg-in-trace '(DQ 3) trace)
                (no-inst-at-stg-in-trace '(IFU) trace))
           (no-INST-at-stg-in-trace '(DQ 2) (step-trace trace MT MA sigs
                                                        ISA spc smc)))))

(defthm no-inst-at-DE2-mt-step-if-not-IFU-DE2-valid
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (DE-valid? (DQ-DE2 (MA-DQ MA)))))
                (not (b1p (IFU-valid? (MA-IFU MA)))))
           (no-INST-at-stg '(DQ 2) (MT-step MT MA sigs)))
  :hints (("goal" :use ((:instance NO-INST-AT-STG-IF-NO-DQ-DE2-VALID)
                        (:instance NO-INST-AT-STG-IF-NO-DQ-DE3-VALID)
                        (:instance NO-INST-AT-IFU-IF-IFU-INVALID))
                :in-theory (enable no-inst-at-stg))))
)

(defthm not-INST-stg-step-inst-DE2-if-not-IFU-DE3
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (b1p (dispatch-inst? MA))
                (not (equal (INST-stg i) '(IFU)))
                (not (equal (INST-stg i) '(DQ 3))))
           (not (equal (INST-stg (step-INST i MT MA sigs)) '(DQ 2))))
  :hints (("goal" :use ((:instance inst-is-at-one-of-the-stages))
                :in-theory (e/d (new-dq-stage coerce-dq-stg
                                   IFU-stg-p DQ-stg-p)
                                 (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-DE2-mt-step-if-not-IFU-DE3-valid-help
```

```
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (no-INST-at-stg-in-trace '(DQ 3) trace)
                  (no-INST-at-stg-in-trace '(IFU) trace))
             (no-INST-at-stg-in-trace '(DQ 2)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(defthm no-inst-at-DE2-mt-step-if-not-IFU-DE3-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA))
                  (not (b1p (DE-valid? (DQ-DE3 (MA-DQ MA)))))
                  (not (b1p (IFU-valid? (MA-IFU MA)))))
             (no-INST-at-stg '(DQ 2) (MT-step MT MA sigs)))
    :hints (("goal" :in-theory (enable no-INST-at-stg)
                    :use ((:instance NO-INST-AT-STG-IF-NO-DQ-DE3-VALID)
                          (:instance NO-INST-AT-IFU-IF-IFU-INVALID)))))
)

(defthm no-inst-at-DE2-mt-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (DE-valid? (DQ-DE2 (MA-DQ (MA-step MA sigs)))))))
             (no-INST-at-stg '(DQ 2) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-dq step-DE2 lift-b-ops
                                     decode-output DE3-out DE2-out))))

;; uniq-inst-at-DE3-mt-step
(defthm not-INST-stg-step-INST-DE3-if-not-dispatch-inst
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (dispatch-inst? MA)))
                  (b1p (DE-valid? (DQ-DE3 (MA-DQ MA))))
                  (not (equal (INST-stg i) '(DQ 3))))
             (not (equal (inst-stg (step-INST i MT MA sigs)) '(DQ 3))))
  :hints (("goal" :use ((:instance inst-is-at-one-of-the-stages))
                  :in-theory (e/d (dq-stg-p new-dq-stage IFU-stg-p
                                    DQ-FULL?
                                    DQ-stg-p coerce-dq-stg)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-DE3-MT-step-if-DE3-valid-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (dispatch-inst? MA)))
                  (b1p (DE-valid? (DQ-DE3 (MA-DQ MA))))
                  (no-inst-at-stg-in-trace '(DQ 3) trace))
             (no-inst-at-stg-in-trace '(DQ 3)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
(defthm uniq-inst-at-DE3-MT-step-if-de3-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
```

```
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (dispatch-inst? MA)))
                (b1p (DE-valid? (DQ-DE3 (MA-DQ MA))))
                (not (b1p (flush-all? MA sigs)))
                (uniq-inst-at-stg-in-trace '(DQ 3) trace))
           (uniq-inst-at-stg-in-trace '(DQ 3)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(defthm uniq-inst-at-DE3-MT-step-if-de3-valid
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (dispatch-inst? MA)))
                (b1p (DE-valid? (DQ-DE3 (MA-DQ MA))))
                (not (b1p (flush-all? MA sigs))))
           (uniq-inst-at-stg '(DQ 3) (MT-step MT MA sigs)))
  :hints (("goal" :use ((:instance UNIQ-INST-AT-STG-IF-DQ-DE3-VALID))
                :in-theory (e/d (uniq-inst-at-stg)
                                (UNIQ-INST-AT-STG-IF-DQ-DE3-VALID)))))
)

(defthm not-INST-stg-step-INST-DE3-if-IFU
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA)
                (or (and (not (b1p (dispatch-inst? MA)))
                         (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))
                         (not (b1p (DE-valid? (DQ-DE3 (MA-DQ MA))))))
                    (and (b1p (dispatch-inst? MA))
                         (b1p (DE-valid? (DQ-DE3 (MA-DQ MA))))))
                (not (equal (INST-stg i) '(IFU))))
           (not (equal (inst-stg (step-INST i MT MA sigs)) '(DQ 3))))
  :hints (("goal" :use ((:instance inst-is-at-one-of-the-stages))
                :in-theory (e/d (dq-stg-p new-dq-stage IFU-stg-p
                                   DQ-stg-p coerce-dq-stg)
                                (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-DE3-MT-step-if-IFU-valid-help-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (dispatch-inst? MA)))
                (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))
                (not (b1p (DE-valid? (DQ-DE3 (MA-DQ MA)))))
                (no-inst-at-stg-in-trace '(IFU) trace))
           (no-inst-at-stg-in-trace '(DQ 3)
                                    (step-trace trace MT MA sigs
                                                ISA spc smc)))))

(local
(defthm uniq-inst-at-DE3-MT-step-if-IFU-valid-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (dispatch-inst? MA)))
                (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))
                (not (b1p (DE-valid? (DQ-DE3 (MA-DQ MA)))))
                (not (b1p (flush-all? MA sigs)))
                (uniq-inst-at-stg-in-trace '(IFU) trace))
           (uniq-inst-at-stg-in-trace '(DQ 3)
```

```
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc)))
  :hints (("goal" :in-theory (enable DQ-FULL? new-dq-stage lift-b-ops)))))

(defthm uniq-inst-at-DE3-MT-step-if-IFU-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (dispatch-inst? MA)))
                  (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))
                  (not (b1p (DE-valid? (DQ-DE3 (MA-DQ MA)))))
                  (b1p (IFU-valid? (MA-IFU MA)))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stg '(DQ 3) (MT-step MT MA sigs)))
    :hints (("goal" :use ((:instance UNIQ-INST-AT-IFU-IF-IFU-VALID))
                  :in-theory (e/d (uniq-inst-at-stg)
                                  (UNIQ-INST-AT-IFU-IF-IFU-VALID)))))
)

(defthm uniq-inst-at-DE3-mt-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (DE-valid? (DQ-DE3 (MA-DQ (MA-step MA sigs))))))
             (uniq-INST-at-stg '(DQ 3) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-dq step-DE3 lift-b-ops
                                     decode-output))))

;; Proof of no-inst-at-DE3-mt-step

(defthm not-INST-stg-step-INST-DE3-if-dispatch-inst
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA)))
             (not (equal (INST-stg (step-INST i MT MA sigs)) '(DQ 3))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (IFU-stg-p new-dq-stage
                                     lift-b-ops DQ-stg-p
                                     DQ-full? coerce-dq-stg)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-DE3-mt-step-if-dispatch-inst-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA)))
             (no-INST-at-stg-in-trace '(DQ 3)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))
  :hints (("goal" :use ((:instance NO-INST-AT-STG-IF-NO-DQ-DE3-VALID))
                  :in-theory (e/d (no-inst-at-stg) ())))))

(defthm no-inst-at-DE3-mt-step-if-dispatch-inst
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA)))
             (no-INST-at-stg '(DQ 3) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (e/d (no-inst-at-stg) ())))
)

(defthm not-INST-stg-step-INST-DE3-if-not-IFU-DE3
```

```
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(DQ 3)))
                  (not (equal (INST-stg i) '(IFU))))
             (not (equal (INST-stg (step-INST i MT MA sigs)) '(DQ 3))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (IFU-stg-p new-dq-stage
                                     step-INST-DQ-INST
                                     step-INST-DQ dispatch-inst
                                     lift-b-ops DQ-stg-p
                                     DQ-full? coerce-dq-stg)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-DE3-mt-step-if-not-IFU-DE3-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-INST-at-stg-in-trace '(DQ 3) trace)
                  (no-INST-at-stg-in-trace '(IFU) trace))
             (no-INST-at-stg-in-trace '(DQ 3)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(defthm no-inst-at-DE3-mt-step-if-not-IFU-DE3-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (DE-valid? (DQ-DE3 (MA-DQ MA)))))
                  (not (b1p (IFU-valid? (MA-IFU MA)))))
             (no-INST-at-stg '(DQ 3) (MT-step MT MA sigs)))
  :hints (("goal" :use ((:instance NO-INST-AT-STG-IF-NO-DQ-DE3-VALID)
                        (:instance NO-INST-AT-IFU-IF-IFU-INVALID))
                  :in-theory (e/d (no-inst-at-stg) ()))))
)

(defthm not-INST-stg-step-INST-DE3-if-not-DE2-valid
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))))
             (not (equal (INST-stg (step-INST i MT MA sigs)) '(DQ 3))))
  :hints (("goal" :use ((:instance inst-is-at-one-of-the-stages)
                        (:instance MT-dq-len-lt-2))
                  :in-theory (e/d (IFU-stg-p new-dq-stage
                                     step-INST-DQ-INST
                                     step-INST-DQ dispatch-inst
                                     lift-b-ops DQ-stg-p
                                     DQ-full? coerce-dq-stg)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-DE3-mt-step-if-not-DE2-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))))
             (no-INST-at-stg-in-trace '(DQ 3)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))
```

```
(defthm no-inst-at-DE3-mt-step-if-not-DE2-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (DE-valid? (DQ-DE2 (MA-DQ MA))))))
             (no-INST-at-stg '(DQ 3) (MT-step MT MA sigs)))
  :hints (("goal" :use ((:instance NO-INST-AT-STG-IF-NO-DQ-DE3-VALID))
                  :in-theory (e/d (no-inst-at-stg) ()))))
)

(defthm no-inst-at-DE3-mt-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (DE-valid? (DQ-DE3 (MA-DQ (MA-step MA sigs)))))))
             (no-INST-at-stg '(DQ 3) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-dq step-DE3 lift-b-ops
                                     decode-output))))

(defthm no-DQ-stg-conflict-preserved
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA))
             (no-DQ-stg-conflict (MT-step MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable no-DQ-stg-conflict))))

(in-theory (disable INST-STG-STEP-IFU-INST-IF-DQ-FULL))

;;; Proof of no-IU-stg-conflict
;; Proof of uniq-inst-at-IU-RS0-MT-step
(in-theory (enable inst-stg-step-inst))

(defthm not-INST-stg-step-INST-IU-RS0-if-not-IU-RS0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (RS-valid? (IU-RS0 (MA-IU MA))))
                  (not (b1p (issue-IU-RS0? (MA-IU MA) MA)))
                  (not (equal (INST-stg i) '(IU RS0))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(IU RS0))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-INST-execute-inst
                                   SELECT-IU-RS0? lift-b-ops
                                   step-inst-execute
                                   step-inst-dq-inst
                                   step-INST-low-level-functions)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-IU-RS0-MT-step-if-RS0-valid-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-INST-at-stg-in-trace '(IU RS0) trace)
                  (b1p (RS-valid? (IU-RS0 (MA-IU MA))))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (issue-IU-RS0? (MA-IU MA) MA))))
             (no-INST-at-stg-in-trace '(IU RS0)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
```

```
(defthm uniq-inst-at-IU-RS0-MT-step-if-RS0-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-INST-at-stg-in-trace '(IU RS0) trace)
                  (b1p (RS-valid? (IU-RS0 (MA-IU MA))))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (issue-IU-RS0? (MA-IU MA) MA))))
             (uniq-INST-at-stg-in-trace '(IU RS0)
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc)))))

(defthm uniq-inst-at-IU-RS0-MT-step-if-RS0-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (RS-valid? (IU-RS0 (MA-IU MA))))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (issue-IU-RS0? (MA-IU MA) MA))))
             (uniq-INST-at-stg '(IU RS0) (MT-step MT MA sigs)))
  :hints (("goal" :use ((:instance UNIQ-INST-AT-IU-RS0-IF-VALID))
                  :in-theory (e/d (uniq-inst-at-stg) ()))))
)

(defthm not-INST-stg-step-INST-IU-RS0-if-not-DE0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (RS-valid? (IU-RS0 (MA-IU MA)))))
                  (not (equal (INST-stg i) '(DQ 0)))
                  (b1p (dispatch-to-IU? MA))
                  (b1p (select-IU-RS0? (MA-IU MA))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(IU RS0))))
    :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                    :in-theory (e/d (step-INST-execute-inst
                                     step-INST-dq-inst
                                     dq-stg-p
                                     step-INST-low-level-functions)
                                    (inst-is-at-one-of-the-stages)))))
(encapsulate nil
(local
(defthm uniq-inst-at-IU-RS0-MT-step-if-dispatch-to-IU-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (RS-valid? (IU-RS0 (MA-IU MA)))))
                  (no-INST-at-stg-in-trace '(DQ 0) trace)
                  (b1p (dispatch-to-IU? MA))
                  (b1p (select-IU-RS0? (MA-IU MA))))
             (no-INST-at-stg-in-trace '(IU RS0)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
(defthm uniq-inst-at-IU-RS0-MT-step-if-dispatch-to-IU-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (RS-valid? (IU-RS0 (MA-IU MA)))))
                  (not (b1p (flush-all? MA sigs)))
                  (uniq-INST-at-stg-in-trace '(DQ 0) trace)
                  (b1p (dispatch-to-IU? MA))
```

```
                (b1p (select-IU-RS0? (MA-IU MA))))
            (uniq-INST-at-stg-in-trace '(IU RS0)
                                       (step-trace trace MT MA sigs
                                                   ISA spc smc)))))

(defthm uniq-inst-at-IU-RS0-MT-step-if-dispatch-to-IU
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (RS-valid? (IU-RS0 (MA-IU MA)))))
                (not (b1p (flush-all? MA sigs)))
                (b1p (dispatch-to-IU? MA))
                (b1p (select-IU-RS0? (MA-IU MA))))
            (uniq-INST-at-stg '(IU RS0) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable uniq-inst-at-stg dispatch-to-IU?
                                     dq-ready-to-iu? lift-b-ops)
                :use (:instance UNIQ-INST-AT-STG-IF-DQ-DE0-VALID))))
)

(defthm uniq-inst-at-IU-RS0-MT-step
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (RS-valid? (IU-RS0 (step-IU MA sigs)))))
            (uniq-inst-at-stg '(IU RS0)
                              (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-IU step-IU-RS0 lift-b-ops))))

;; Proof of no-inst-at-IU-RS0-MT-step
(defthm not-INST-stg-step-INST-if-issue-IU-RS0
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (RS-valid? (IU-RS0 (MA-IU MA))))
                (b1p (issue-IU-RS0? (MA-IU MA) MA)))
            (not (equal (INST-stg (step-INST i MT MA sigs))
                        '(IU RS0))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                :in-theory (e/d (step-inst-execute-inst
                                   step-inst-dq-inst
                                   dq-stg-p issue-IU-Rs0?
                                   select-IU-RS0? lift-b-ops
                                   step-INST-low-level-functions)
                                (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-IU-RS0-MT-step-if-issue-IU-RS0-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (RS-valid? (IU-RS0 (MA-IU MA))))
                (b1p (issue-IU-RS0? (MA-IU MA) MA)))
            (no-INST-at-stg-in-trace '(IU RS0)
                                     (step-trace trace MT MA sigs
                                                 ISA spc smc)))))

(defthm no-inst-at-IU-RS0-MT-step-if-issue-IU-RS0
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (RS-valid? (IU-RS0 (MA-IU MA))))
                (b1p (issue-IU-RS0? (MA-IU MA) MA)))
            (no-INST-at-stg '(IU RS0) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg))))
```

```
)

(defthm not-INST-stg-step-inst-IU-RS0-if-not-dispatch-to-IU
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(IU RS0)))
                  (not (b1p (dispatch-to-IU? MA))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(IU RS0))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-INST-execute-INST
                                   step-INST-low-level-functions
                                   step-INST-DQ-inst)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-IU-RS0-MT-step-if-not-dispatch-to-IU-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(IU RS0) trace)
                  (not (b1p (dispatch-to-IU? MA))))
             (no-INST-at-stg-in-trace '(IU RS0)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(defthm no-inst-at-IU-RS0-MT-step-if-not-dispatch-to-IU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (RS-valid? (IU-RS0 (MA-IU MA)))))
                  (not (b1p (dispatch-to-IU? MA))))
             (no-INST-at-stg '(IU RS0) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg)
                  :use (:instance no-inst-at-IU-RS0))))
)

(defthm no-inst-at-IU-RS0-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (RS-valid? (IU-RS0 (step-IU MA sigs))))))
             (no-inst-at-stg '(IU RS0)
                             (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-IU step-IU-RS0 lift-b-ops
                                     select-IU-RS0?))))

;; Proof of uniq-inst-at-IU-RS1-MT-step
(defthm not-INST-stg-step-INST-IU-RS1-if-not-IU-RS1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (RS-valid? (IU-RS1 (MA-IU MA))))
                  (not (b1p (issue-IU-RS1? (MA-IU MA) MA)))
                  (not (equal (INST-stg i) '(IU RS1))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(IU RS1))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-INST-execute-inst
                                   SELECT-IU-RS1? lift-b-ops
                                   select-IU-RS0? dq-stg-p
```

```
                                    IU-ready? dispatch-to-IU?
                                    step-inst-execute
                                    step-inst-dq-inst
                                    step-INST-low-level-functions)
                                  (inst-is-at-one-of-the-stages)))))
(encapsulate nil
(local
(defthm uniq-inst-at-IU-RS1-MT-step-if-RS1-valid-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-INST-at-stg-in-trace '(IU RS1) trace)
                  (b1p (RS-valid? (IU-RS1 (MA-IU MA))))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (issue-IU-RS1? (MA-IU MA) MA))))
             (no-INST-at-stg-in-trace '(IU RS1)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
(defthm uniq-inst-at-IU-RS1-MT-step-if-RS1-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-INST-at-stg-in-trace '(IU RS1) trace)
                  (b1p (RS-valid? (IU-RS1 (MA-IU MA))))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (issue-IU-RS1? (MA-IU MA) MA))))
             (uniq-INST-at-stg-in-trace '(IU RS1)
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc)))))

(defthm uniq-inst-at-IU-RS1-MT-step-if-RS1-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (RS-valid? (IU-RS1 (MA-IU MA))))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (issue-IU-RS1? (MA-IU MA) MA))))
             (uniq-INST-at-stg '(IU RS1) (MT-step MT MA sigs)))
  :hints (("goal" :use ((:instance UNIQ-INST-AT-IU-RS1-IF-VALID))
                  :in-theory (e/d (uniq-inst-at-stg) ()))))
)

(defthm not-INST-stg-step-INST-IU-RS1-if-not-DE1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (RS-valid? (IU-RS1 (MA-IU MA)))))
                  (not (equal (INST-stg i) '(DQ 0)))
                  (b1p (dispatch-to-IU? MA))
                  (b1p (select-IU-RS1? (MA-IU MA))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(IU RS1))))
    :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                    :in-theory (e/d (step-INST-execute-inst
                                     step-INST-dq-inst
                                     dq-stg-p
                                     step-INST-low-level-functions)
                                    (inst-is-at-one-of-the-stages)))))
(encapsulate nil
(local
(defthm uniq-inst-at-IU-RS1-MT-step-if-dispatch-to-IU-help-help
```

```
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (RS-valid? (IU-RS1 (MA-IU MA)))))
                  (no-INST-at-stg-in-trace '(DQ 0) trace)
                  (b1p (dispatch-to-IU? MA))
                  (b1p (select-IU-RS1? (MA-IU MA))))
             (no-INST-at-stg-in-trace '(IU RS1)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
(defthm uniq-inst-at-IU-RS1-MT-step-if-dispatch-to-IU-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (RS-valid? (IU-RS1 (MA-IU MA)))))
                  (not (b1p (flush-all? MA sigs)))
                  (uniq-INST-at-stg-in-trace '(DQ 0) trace)
                  (b1p (dispatch-to-IU? MA))
                  (b1p (select-IU-RS1? (MA-IU MA))))
             (uniq-INST-at-stg-in-trace '(IU RS1)
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc)))
  :hints (("goal" :in-theory (enable lift-b-ops select-IU-RS1?
                                     select-IU-RS0?)))))

(defthm uniq-inst-at-IU-RS1-MT-step-if-dispatch-to-IU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (RS-valid? (IU-RS1 (MA-IU MA)))))
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (dispatch-to-IU? MA))
                  (b1p (select-IU-RS1? (MA-IU MA))))
             (uniq-INST-at-stg '(IU RS1) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable uniq-inst-at-stg dispatch-to-IU?
                                     dq-ready-to-iu? lift-b-ops)
                  :use (:instance UNIQ-INST-AT-STG-IF-DQ-DE0-VALID))))
)

(defthm uniq-inst-at-IU-RS1-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (RS-valid? (IU-RS1 (step-IU MA sigs)))))
             (uniq-inst-at-stg '(IU RS1)
                               (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-IU step-IU-RS1
                                     lift-b-ops))))

;; Proof of no-inst-at-IU-RS1-MT-step
(defthm not-INST-stg-step-INST-if-issue-IU-RS1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (RS-valid? (IU-RS1 (MA-IU MA))))
                  (b1p (issue-IU-RS1? (MA-IU MA) MA)))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(IU RS1))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-inst-execute-inst
                                     step-inst-dq-inst
                                     dq-stg-p issue-IU-RS1?
```

```
                                   DISPATCH-TO-IU? IU-ready?
                                   select-IU-RS0? lift-b-ops
                                   step-INST-low-level-functions)
                                (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-IU-RS1-MT-step-if-issue-IU-RS1-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (RS-valid? (IU-RS1 (MA-IU MA))))
                  (b1p (issue-IU-RS1? (MA-IU MA) MA)))
             (no-INST-at-stg-in-trace '(IU RS1)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(defthm no-inst-at-IU-RS1-MT-step-if-issue-IU-RS1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (RS-valid? (IU-RS1 (MA-IU MA))))
                  (b1p (issue-IU-RS1? (MA-IU MA) MA)))
             (no-INST-at-stg '(IU RS1) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg))))
)

(defthm not-INST-stg-step-inst-IU-RS1-if-not-dispatch-to-IU
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(IU RS1)))
                  (or (not (b1p (dispatch-to-IU? MA)))
                      (not (b1p (select-IU-RS1? (MA-IU MA))))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(IU RS1))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-INST-execute-INST
                                   select-IU-RS1? select-IU-RS0?
                                   lift-b-ops dq-stg-p
                                   DISPATCH-TO-IU? IU-ready?
                                   step-INST-low-level-functions
                                   step-INST-DQ-inst)
                                (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-IU-RS1-MT-step-if-not-dispatch-to-IU-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(IU RS1) trace)
                  (or (not (b1p (dispatch-to-IU? MA)))
                      (not (b1p (select-IU-RS1? (MA-IU MA))))))
             (no-INST-at-stg-in-trace '(IU RS1)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(defthm no-inst-at-IU-RS1-MT-step-if-not-dispatch-to-IU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (RS-valid? (IU-RS1 (MA-IU MA)))))
                  (or (not (b1p (dispatch-to-IU? MA)))
```

```
                    (not (b1p (select-IU-RS1? (MA-IU MA))))))
             (no-INST-at-stg '(IU RS1) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg)
                  :use (:instance no-inst-at-IU-RS1))))
)

(defthm no-inst-at-IU-RS1-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (RS-valid? (IU-RS1 (step-IU MA sigs))))))
             (no-inst-at-stg '(IU RS1) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-IU step-IU-RS1
                                     lift-b-ops))))

(defthm no-IU-stg-conflict-preserved
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA))
             (no-IU-stg-conflict (MT-step MT MA sigs) (MA-step MA sigs)))
    :hints (("goal" :in-theory (enable no-IU-stg-conflict))))

;;; Proof of no-BU-stg-conflict-preserved
;; Proof of uniq-inst-at-BU-RS0-MT-step
(defthm not-INST-stg-step-INST-BU-RS0-if-not-BU-RS0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (BU-RS-valid? (BU-RS0 (MA-BU MA))))
                  (not (b1p (issue-BU-RS0? (MA-BU MA) MA)))
                  (not (equal (INST-stg i) '(BU RS0))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(BU RS0))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-INST-execute-inst
                                     SELECT-BU-RS0? lift-b-ops
                                     step-inst-execute
                                     step-inst-dq-inst
                                     step-INST-low-level-functions)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-BU-RS0-MT-step-if-RS0-valid-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-INST-at-stg-in-trace '(BU RS0) trace)
                  (b1p (BU-RS-valid? (BU-RS0 (MA-BU MA))))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (issue-BU-RS0? (MA-BU MA) MA))))
             (no-INST-at-stg-in-trace '(BU RS0)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
(defthm uniq-inst-at-BU-RS0-MT-step-if-RS0-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-INST-at-stg-in-trace '(BU RS0) trace)
                  (b1p (BU-RS-valid? (BU-RS0 (MA-BU MA))))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (issue-BU-RS0? (MA-BU MA) MA))))
```

```
            (uniq-INST-at-stg-in-trace '(BU RS0)
                                       (step-trace trace MT MA sigs
                                                   ISA spc smc)))))

(defthm uniq-inst-at-BU-RS0-MT-step-if-RS0-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (BU-RS-valid? (BU-RS0 (MA-BU MA))))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (issue-BU-RS0? (MA-BU MA) MA))))
             (uniq-INST-at-stg '(BU RS0) (MT-step MT MA sigs)))
  :hints (("goal" :use ((:instance UNIQ-INST-AT-BU-RS0-IF-VALID))
                  :in-theory (e/d (uniq-inst-at-stg) ()))))
)

(defthm not-INST-stg-step-INST-BU-RS0-if-not-DE0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (BU-RS-valid? (BU-RS0 (MA-BU MA)))))
                  (not (equal (INST-stg i) '(DQ 0)))
                  (b1p (dispatch-to-BU? MA))
                  (b1p (select-BU-RS0? (MA-BU MA))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(BU RS0))))
    :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                    :in-theory (e/d (step-INST-execute-inst
                                       step-INST-dq-inst
                                       dq-stg-p
                                       step-INST-low-level-functions)
                                    (inst-is-at-one-of-the-stages)))))
(encapsulate nil
(local
(defthm uniq-inst-at-BU-RS0-MT-step-if-dispatch-to-BU-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (BU-RS-valid? (BU-RS0 (MA-BU MA)))))
                  (no-INST-at-stg-in-trace '(DQ 0) trace)
                  (b1p (dispatch-to-BU? MA))
                  (b1p (select-BU-RS0? (MA-BU MA))))
             (no-INST-at-stg-in-trace '(BU RS0)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
(defthm uniq-inst-at-BU-RS0-MT-step-if-dispatch-to-BU-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (BU-RS-valid? (BU-RS0 (MA-BU MA)))))
                  (not (b1p (flush-all? MA sigs)))
                  (uniq-INST-at-stg-in-trace '(DQ 0) trace)
                  (b1p (dispatch-to-BU? MA))
                  (b1p (select-BU-RS0? (MA-BU MA))))
             (uniq-INST-at-stg-in-trace '(BU RS0)
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc)))))

(defthm uniq-inst-at-BU-RS0-MT-step-if-dispatch-to-BU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
```

```
                (not (b1p (BU-RS-valid? (BU-RS0 (MA-BU MA)))))
                (not (b1p (flush-all? MA sigs)))
                (b1p (dispatch-to-BU? MA))
                (b1p (select-BU-RS0? (MA-BU MA))))
           (uniq-INST-at-stg '(BU RS0) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable uniq-inst-at-stg dispatch-to-BU?
                                     dq-ready-to-bu? lift-b-ops)
                :use (:instance UNIQ-INST-AT-STG-IF-DQ-DE0-VALID))))
)

(defthm uniq-inst-at-BU-RS0-MT-step
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (BU-RS-valid? (BU-RS0 (step-BU MA sigs)))))
           (uniq-inst-at-stg '(BU RS0) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-BU step-BU-RS0
                                     lift-b-ops))))

;; Proof of no-inst-at-BU-RS0-MT-step
(defthm not-INST-stg-step-INST-if-issue-BU-RS0
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (BU-RS-valid? (BU-RS0 (MA-BU MA))))
                (b1p (issue-BU-RS0? (MA-BU MA) MA)))
           (not (equal (INST-stg (step-INST i MT MA sigs))
                       '(BU RS0))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                :in-theory (e/d (step-inst-execute-inst
                                   step-inst-dq-inst
                                   dq-stg-p issue-BU-Rs0?
                                   select-BU-RS0? lift-b-ops
                                   step-INST-low-level-functions)
                                (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-BU-RS0-MT-step-if-issue-BU-RS0-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (BU-RS-valid? (BU-RS0 (MA-BU MA))))
                (b1p (issue-BU-RS0? (MA-BU MA) MA)))
           (no-INST-at-stg-in-trace '(BU RS0)
                                    (step-trace trace MT MA sigs
                                                ISA spc smc)))))

(defthm no-inst-at-BU-RS0-MT-step-if-issue-BU-RS0
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (BU-RS-valid? (BU-RS0 (MA-BU MA))))
                (b1p (issue-BU-RS0? (MA-BU MA) MA)))
           (no-INST-at-stg '(BU RS0) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg))))

)

(defthm not-INST-stg-step-inst-BU-RS0-if-not-dispatch-to-BU
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA)
                (not (equal (INST-stg i) '(BU RS0)))
```

```
                (not (b1p (dispatch-to-BU? MA))))
            (not (equal (INST-stg (step-INST i MT MA sigs))
                        '(BU RS0))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-INST-execute-INST
                                   step-INST-low-level-functions
                                   step-INST-DQ-inst)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-BU-RS0-MT-step-if-not-dispatch-to-BU-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(BU RS0) trace)
                  (not (b1p (dispatch-to-BU? MA))))
             (no-INST-at-stg-in-trace '(BU RS0)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(defthm no-inst-at-BU-RS0-MT-step-if-not-dispatch-to-BU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (BU-RS-valid? (BU-RS0 (MA-BU MA)))))
                  (not (b1p (dispatch-to-BU? MA))))
             (no-INST-at-stg '(BU RS0) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg)
                  :use (:instance no-inst-at-BU-RS0))))
)

(defthm no-inst-at-BU-RS0-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (BU-RS-valid? (BU-RS0 (step-BU MA sigs))))))
             (no-inst-at-stg '(BU RS0) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-BU step-BU-RS0 lift-b-ops
                                     select-BU-RS0?))))

;; Proof of uniq-inst-at-BU-RS1-MT-step
(defthm not-INST-stg-step-INST-BU-RS1-if-not-BU-RS1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (BU-RS-valid? (BU-RS1 (MA-BU MA))))
                  (not (b1p (issue-BU-RS1? (MA-BU MA) MA)))
                  (not (equal (INST-stg i) '(BU RS1))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(BU RS1))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-INST-execute-inst
                                   SELECT-BU-RS1? lift-b-ops
                                   select-BU-RS0? dq-stg-p
                                   BU-ready? dispatch-to-BU?
                                   step-inst-execute
                                   step-inst-dq-inst
                                   step-INST-low-level-functions)
                                  (inst-is-at-one-of-the-stages)))))
(encapsulate nil
(local
(defthm uniq-inst-at-BU-RS1-MT-step-if-RS1-valid-help-help
    (implies (and (inv MT MA)
```

```
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (no-INST-at-stg-in-trace '(BU RS1) trace)
                (b1p (BU-RS-valid? (BU-RS1 (MA-BU MA))))
                (not (b1p (flush-all? MA sigs)))
                (not (b1p (issue-BU-RS1? (MA-BU MA) MA))))
           (no-INST-at-stg-in-trace '(BU RS1)
                                    (step-trace trace MT MA sigs
                                                ISA spc smc)))))

(local
(defthm uniq-inst-at-BU-RS1-MT-step-if-RS1-valid-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (uniq-INST-at-stg-in-trace '(BU RS1) trace)
                (b1p (BU-RS-valid? (BU-RS1 (MA-BU MA))))
                (not (b1p (flush-all? MA sigs)))
                (not (b1p (issue-BU-RS1? (MA-BU MA) MA))))
           (uniq-INST-at-stg-in-trace '(BU RS1)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(defthm uniq-inst-at-BU-RS1-MT-step-if-RS1-valid
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (BU-RS-valid? (BU-RS1 (MA-BU MA))))
                (not (b1p (flush-all? MA sigs)))
                (not (b1p (issue-BU-RS1? (MA-BU MA) MA))))
           (uniq-INST-at-stg '(BU RS1) (MT-step MT MA sigs)))
  :hints (("goal" :use ((:instance UNIQ-INST-AT-BU-RS1-IF-VALID))
                :in-theory (e/d (uniq-inst-at-stg) ()))))
)

(defthm not-INST-stg-step-INST-BU-RS1-if-not-DE1
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (BU-RS-valid? (BU-RS1 (MA-BU MA)))))
                (not (equal (INST-stg i) '(DQ 0)))
                (b1p (dispatch-to-BU? MA))
                (b1p (select-BU-RS1? (MA-BU MA))))
           (not (equal (INST-stg (step-INST i MT MA sigs))
                       '(BU RS1))))
    :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                    :in-theory (e/d (step-INST-execute-inst
                                     step-INST-dq-inst
                                     dq-stg-p
                                     step-INST-low-level-functions)
                                    (inst-is-at-one-of-the-stages)))))
(encapsulate nil
(local
(defthm uniq-inst-at-BU-RS1-MT-step-if-dispatch-to-BU-help-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (BU-RS-valid? (BU-RS1 (MA-BU MA)))))
                (no-INST-at-stg-in-trace '(DQ 0) trace)
                (b1p (dispatch-to-BU? MA))
                (b1p (select-BU-RS1? (MA-BU MA))))
           (no-INST-at-stg-in-trace '(BU RS1)
                                    (step-trace trace MT MA sigs
```

```
                                                ISA spc smc)))))

(local
(defthm uniq-inst-at-BU-RS1-MT-step-if-dispatch-to-BU-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (BU-RS-valid? (BU-RS1 (MA-BU MA)))))
                  (not (b1p (flush-all? MA sigs)))
                  (uniq-INST-at-stg-in-trace '(DQ 0) trace)
                  (b1p (dispatch-to-BU? MA))
                  (b1p (select-BU-RS1? (MA-BU MA))))
             (uniq-INST-at-stg-in-trace '(BU RS1)
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc)))
  :hints (("goal" :in-theory (enable lift-b-ops select-BU-RS1?
                                     select-BU-RS0?)))))

(defthm uniq-inst-at-BU-RS1-MT-step-if-dispatch-to-BU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (BU-RS-valid? (BU-RS1 (MA-BU MA)))))
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (dispatch-to-BU? MA))
                  (b1p (select-BU-RS1? (MA-BU MA))))
             (uniq-INST-at-stg '(BU RS1) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable uniq-inst-at-stg dispatch-to-BU?
                                     dq-ready-to-bu? lift-b-ops)
                  :use (:instance UNIQ-INST-AT-STG-IF-DQ-DE0-VALID))))
)

(defthm uniq-inst-at-BU-RS1-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (BU-RS-valid? (BU-RS1 (step-BU MA sigs)))))
             (uniq-inst-at-stg '(BU RS1) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable lift-b-ops step-BU step-BU-RS1))))

;; Proof of no-inst-at-BU-RS1-MT-step
(defthm not-INST-stg-step-INST-if-issue-BU-RS1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (BU-RS-valid? (BU-RS1 (MA-BU MA))))
                  (b1p (issue-BU-RS1? (MA-BU MA) MA)))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(BU RS1))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-inst-execute-inst
                                   step-inst-dq-inst
                                   dq-stg-p issue-BU-RS1?
                                   DISPATCH-TO-BU? BU-ready?
                                   select-BU-RS0? lift-b-ops
                                   step-INST-low-level-functions)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-BU-RS1-MT-step-if-issue-BU-RS1-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
```

```
                (b1p (BU-RS-valid? (BU-RS1 (MA-BU MA))))
                (b1p (issue-BU-RS1? (MA-BU MA) MA)))
           (no-INST-at-stg-in-trace '(BU RS1)
                                    (step-trace trace MT MA sigs
                                                ISA spc smc)))))

(defthm no-inst-at-BU-RS1-MT-step-if-issue-BU-RS1
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (BU-RS-valid? (BU-RS1 (MA-BU MA))))
                (b1p (issue-BU-RS1? (MA-BU MA) MA)))
           (no-INST-at-stg '(BU RS1) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg))))
)

(defthm not-INST-stg-step-inst-BU-RS1-if-not-dispatch-to-BU
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA)
                (not (equal (INST-stg i) '(BU RS1)))
                (or (not (b1p (dispatch-to-BU? MA)))
                    (not (b1p (select-BU-RS1? (MA-BU MA))))))
           (not (equal (INST-stg (step-INST i MT MA sigs))
                       '(BU RS1))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                :in-theory (e/d (step-INST-execute-INST
                                   select-BU-RS1? select-BU-RS0?
                                   lift-b-ops dq-stg-p
                                   DISPATCH-TO-BU? BU-ready?
                                   step-INST-low-level-functions
                                   step-INST-DQ-inst)
                                (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-BU-RS1-MT-step-if-not-dispatch-to-BU-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (no-inst-at-stg-in-trace '(BU RS1) trace)
                (or (not (b1p (dispatch-to-BU? MA)))
                    (not (b1p (select-BU-RS1? (MA-BU MA))))))
           (no-INST-at-stg-in-trace '(BU RS1)
                                    (step-trace trace MT MA sigs
                                                ISA spc smc)))))

(defthm no-inst-at-BU-RS1-MT-step-if-not-dispatch-to-BU
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (BU-RS-valid? (BU-RS1 (MA-BU MA)))))
                (or (not (b1p (dispatch-to-BU? MA)))
                    (not (b1p (select-BU-RS1? (MA-BU MA))))))
           (no-INST-at-stg '(BU RS1) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg)
                :use (:instance no-inst-at-BU-RS1))))
)

(defthm no-inst-at-BU-RS1-MT-step
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (BU-RS-valid? (BU-RS1 (step-BU MA sigs))))))
           (no-inst-at-stg '(BU RS1) (MT-step MT MA sigs)))
```

```
  :hints (("goal" :in-theory (enable step-BU step-BU-RS1 lift-b-ops))))

(defthm no-BU-stg-conflict-preserved
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA))
             (no-BU-stg-conflict (MT-step MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable no-BU-stg-conflict))))

;;; Proof of no-MU-stg-conflict-preserved
;; Proof of uniq-INST-at-MU-RS0-MT-step
(defthm not-INST-stg-step-INST-MU-RS0-if-not-MU-RS0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (RS-valid? (MU-RS0 (MA-MU MA))))
                  (not (b1p (issue-MU-RS0? (MA-MU MA) MA)))
                  (not (equal (INST-stg i) '(MU RS0))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(MU RS0))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-INST-execute-inst
                                   SELECT-MU-RS0? lift-b-ops
                                   step-inst-execute
                                   step-inst-dq-inst
                                   step-INST-low-level-functions)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-MU-RS0-MT-step-if-RS0-valid-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-INST-at-stg-in-trace '(MU RS0) trace)
                  (b1p (RS-valid? (MU-RS0 (MA-MU MA))))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (issue-MU-RS0? (MA-MU MA) MA))))
             (no-INST-at-stg-in-trace '(MU RS0)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
(defthm uniq-inst-at-MU-RS0-MT-step-if-RS0-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-INST-at-stg-in-trace '(MU RS0) trace)
                  (b1p (RS-valid? (MU-RS0 (MA-MU MA))))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (issue-MU-RS0? (MA-MU MA) MA))))
             (uniq-INST-at-stg-in-trace '(MU RS0)
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc)))))

(defthm uniq-inst-at-MU-RS0-MT-step-if-RS0-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (RS-valid? (MU-RS0 (MA-MU MA))))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (issue-MU-RS0? (MA-MU MA) MA))))
             (uniq-INST-at-stg '(MU RS0) (MT-step MT MA sigs)))
  :hints (("goal" :use ((:instance UNIQ-INST-AT-MU-RS0-IF-VALID))
```

```
                :in-theory (e/d (uniq-inst-at-stg) ())))
)

(defthm not-INST-stg-step-INST-MU-RS0-if-not-DE0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (RS-valid? (MU-RS0 (MA-MU MA)))))
                  (not (equal (INST-stg i) '(DQ 0)))
                  (b1p (dispatch-to-MU? MA))
                  (b1p (select-MU-RS0? (MA-MU MA))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(MU RS0))))
    :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                    :in-theory (e/d (step-INST-execute-inst
                                     step-INST-dq-inst
                                     dq-stg-p
                                     step-INST-low-level-functions)
                                    (inst-is-at-one-of-the-stages)))))
(encapsulate nil
(local
(defthm uniq-inst-at-MU-RS0-MT-step-if-dispatch-to-MU-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (RS-valid? (MU-RS0 (MA-MU MA)))))
                  (no-INST-at-stg-in-trace '(DQ 0) trace)
                  (b1p (dispatch-to-MU? MA))
                  (b1p (select-MU-RS0? (MA-MU MA))))
             (no-INST-at-stg-in-trace '(MU RS0)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
(defthm uniq-inst-at-MU-RS0-MT-step-if-dispatch-to-MU-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (RS-valid? (MU-RS0 (MA-MU MA)))))
                  (not (b1p (flush-all? MA sigs)))
                  (uniq-INST-at-stg-in-trace '(DQ 0) trace)
                  (b1p (dispatch-to-MU? MA))
                  (b1p (select-MU-RS0? (MA-MU MA))))
             (uniq-INST-at-stg-in-trace '(MU RS0)
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc)))))

(defthm uniq-inst-at-MU-RS0-MT-step-if-dispatch-to-MU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (RS-valid? (MU-RS0 (MA-MU MA)))))
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (dispatch-to-MU? MA))
                  (b1p (select-MU-RS0? (MA-MU MA))))
             (uniq-INST-at-stg '(MU RS0) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable uniq-inst-at-stg dispatch-to-MU?
                                     dq-ready-to-mu? lift-b-ops)
                  :use (:instance UNIQ-INST-AT-STG-IF-DQ-DE0-VALID))))
)

(defthm uniq-INST-at-MU-RS0-MT-step
    (implies (and (inv MT MA)
```

```
                (MAETT-p MT) (MA-state-p MA)
                (b1p (RS-valid? (MU-RS0 (step-MU MA sigs)))))
           (uniq-inst-at-stg '(MU RS0) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-MU step-MU-RS0 lift-b-ops))))

;; Proof of no-INST-at-MU-RS0-MT-step
(defthm not-INST-stg-step-INST-if-issue-MU-RS0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (RS-valid? (MU-RS0 (MA-MU MA))))
                  (b1p (issue-MU-RS0? (MA-MU MA) MA)))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(MU RS0))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-inst-execute-inst
                                     step-inst-dq-inst
                                     dq-stg-p issue-MU-Rs0?
                                     select-MU-RS0? lift-b-ops
                                     step-INST-low-level-functions)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-MU-RS0-MT-step-if-issue-MU-RS0-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (RS-valid? (MU-RS0 (MA-MU MA))))
                  (b1p (issue-MU-RS0? (MA-MU MA) MA)))
             (no-INST-at-stg-in-trace '(MU RS0)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(defthm no-inst-at-MU-RS0-MT-step-if-issue-MU-RS0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (RS-valid? (MU-RS0 (MA-MU MA))))
                  (b1p (issue-MU-RS0? (MA-MU MA) MA)))
             (no-INST-at-stg '(MU RS0) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg))))

)

(defthm not-INST-stg-step-inst-MU-RS0-if-not-dispatch-to-MU
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(MU RS0)))
                  (not (b1p (dispatch-to-MU? MA))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(MU RS0))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-INST-execute-INST
                                     step-INST-low-level-functions
                                     step-INST-DQ-inst)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-MU-RS0-MT-step-if-not-dispatch-to-MU-help
    (implies (and (inv MT MA)
```

```
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (no-inst-at-stg-in-trace '(MU RS0) trace)
                (not (b1p (dispatch-to-MU? MA))))
           (no-INST-at-stg-in-trace '(MU RS0)
                                    (step-trace trace MT MA sigs
                                                ISA spc smc)))))

(defthm no-inst-at-MU-RS0-MT-step-if-not-dispatch-to-MU
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (RS-valid? (MU-RS0 (MA-MU MA)))))
                (not (b1p (dispatch-to-MU? MA))))
           (no-INST-at-stg '(MU RS0) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg)
                :use (:instance no-inst-at-MU-RS0))))
)

(defthm no-INST-at-MU-RS0-MT-step
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (RS-valid? (MU-RS0 (step-MU MA sigs))))))
           (no-inst-at-stg '(MU RS0) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-MU step-MU-RS0 lift-b-ops
                                     select-MU-RS0?))))

;; Proof of uniq-INST-at-MU-RS1-MT-step
(defthm not-INST-stg-step-INST-MU-RS1-if-not-MU-RS1
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (RS-valid? (MU-RS1 (MA-MU MA))))
                (not (b1p (issue-MU-RS1? (MA-MU MA) MA)))
                (not (equal (INST-stg i) '(MU RS1))))
           (not (equal (INST-stg (step-INST i MT MA sigs))
                       '(MU RS1))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                :in-theory (e/d (step-INST-execute-inst
                                   SELECT-MU-RS1? lift-b-ops
                                   select-MU-RS0? dq-stg-p
                                   MU-ready? dispatch-to-MU?
                                   step-inst-execute
                                   step-inst-dq-inst
                                   step-INST-low-level-functions)
                                 (inst-is-at-one-of-the-stages)))))
(encapsulate nil
(local
(defthm uniq-inst-at-MU-RS1-MT-step-if-RS1-valid-help-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (no-INST-at-stg-in-trace '(MU RS1) trace)
                (b1p (RS-valid? (MU-RS1 (MA-MU MA))))
                (not (b1p (flush-all? MA sigs)))
                (not (b1p (issue-MU-RS1? (MA-MU MA) MA))))
           (no-INST-at-stg-in-trace '(MU RS1)
                                    (step-trace trace MT MA sigs
                                                ISA spc smc)))))

(local
(defthm uniq-inst-at-MU-RS1-MT-step-if-RS1-valid-help
    (implies (and (inv MT MA)
```

```
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (uniq-INST-at-stg-in-trace '(MU RS1) trace)
                (b1p (RS-valid? (MU-RS1 (MA-MU MA))))
                (not (b1p (flush-all? MA sigs)))
                (not (b1p (issue-MU-RS1? (MA-MU MA) MA))))
           (uniq-INST-at-stg-in-trace '(MU RS1)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(defthm uniq-inst-at-MU-RS1-MT-step-if-RS1-valid
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (RS-valid? (MU-RS1 (MA-MU MA))))
                (not (b1p (flush-all? MA sigs)))
                (not (b1p (issue-MU-RS1? (MA-MU MA) MA))))
           (uniq-INST-at-stg '(MU RS1) (MT-step MT MA sigs)))
  :hints (("goal" :use ((:instance UNIQ-INST-AT-MU-RS1-IF-VALID))
                :in-theory (e/d (uniq-inst-at-stg) ()))))
)

(defthm not-INST-stg-step-INST-MU-RS1-if-not-DE1
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (RS-valid? (MU-RS1 (MA-MU MA)))))
                (not (equal (INST-stg i) '(DQ 0)))
                (b1p (dispatch-to-MU? MA))
                (b1p (select-MU-RS1? (MA-MU MA))))
           (not (equal (INST-stg (step-INST i MT MA sigs))
                       '(MU RS1))))
    :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                    :in-theory (e/d (step-INST-execute-inst
                                     step-INST-dq-inst
                                     dq-stg-p
                                     step-INST-low-level-functions)
                                    (inst-is-at-one-of-the-stages)))))
(encapsulate nil
(local
(defthm uniq-inst-at-MU-RS1-MT-step-if-dispatch-to-MU-help-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (RS-valid? (MU-RS1 (MA-MU MA)))))
                (no-INST-at-stg-in-trace '(DQ 0) trace)
                (b1p (dispatch-to-MU? MA))
                (b1p (select-MU-RS1? (MA-MU MA))))
           (no-INST-at-stg-in-trace '(MU RS1)
                                    (step-trace trace MT MA sigs
                                                ISA spc smc)))))

(local
(defthm uniq-inst-at-MU-RS1-MT-step-if-dispatch-to-MU-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (RS-valid? (MU-RS1 (MA-MU MA)))))
                (not (b1p (flush-all? MA sigs)))
                (uniq-INST-at-stg-in-trace '(DQ 0) trace)
                (b1p (dispatch-to-MU? MA))
                (b1p (select-MU-RS1? (MA-MU MA))))
           (uniq-INST-at-stg-in-trace '(MU RS1)
```

```
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc)))
  :hints (("goal" :in-theory (enable lift-b-ops select-MU-RS1?
                                     select-MU-RS0?)))))

(defthm uniq-inst-at-MU-RS1-MT-step-if-dispatch-to-MU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (RS-valid? (MU-RS1 (MA-MU MA)))))
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (dispatch-to-MU? MA))
                  (b1p (select-MU-RS1? (MA-MU MA))))
             (uniq-INST-at-stg '(MU RS1) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable uniq-inst-at-stg dispatch-to-MU?
                                     dq-ready-to-mu? lift-b-ops)
                  :use (:instance UNIQ-INST-AT-STG-IF-DQ-DEO-VALID))))
)

(defthm uniq-INST-at-MU-RS1-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (RS-valid? (MU-RS1 (step-MU MA sigs)))))
             (uniq-inst-at-stg '(MU RS1) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-MU step-MU-RS1 lift-b-ops))))

;; Proof of no-INST-at-MU-RS1-MT-step
(defthm not-INST-stg-step-INST-if-issue-MU-RS1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (RS-valid? (MU-RS1 (MA-MU MA))))
                  (b1p (issue-MU-RS1? (MA-MU MA) MA)))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(MU RS1))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-inst-execute-inst
                                     step-inst-dq-inst
                                     dq-stg-p issue-MU-RS1?
                                     DISPATCH-TO-MU? MU-ready?
                                     select-MU-RS0? lift-b-ops
                                     step-INST-low-level-functions)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-MU-RS1-MT-step-if-issue-MU-RS1-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (RS-valid? (MU-RS1 (MA-MU MA))))
                  (b1p (issue-MU-RS1? (MA-MU MA) MA)))
             (no-INST-at-stg-in-trace '(MU RS1)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(defthm no-inst-at-MU-RS1-MT-step-if-issue-MU-RS1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (RS-valid? (MU-RS1 (MA-MU MA))))
                  (b1p (issue-MU-RS1? (MA-MU MA) MA)))
             (no-INST-at-stg '(MU RS1) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg))))
```

```
)

(defthm not-INST-stg-step-inst-MU-RS1-if-not-dispatch-to-MU
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(MU RS1)))
                  (or (not (b1p (dispatch-to-MU? MA)))
                      (not (b1p (select-MU-RS1? (MA-MU MA))))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(MU RS1))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-INST-execute-INST
                                   select-MU-RS1? select-MU-RS0?
                                   lift-b-ops dq-stg-p
                                   DISPATCH-TO-MU? MU-ready?
                                   step-INST-low-level-functions
                                   step-INST-DQ-inst)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-MU-RS1-MT-step-if-not-dispatch-to-MU-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(MU RS1) trace)
                  (or (not (b1p (dispatch-to-MU? MA)))
                      (not (b1p (select-MU-RS1? (MA-MU MA))))))
             (no-INST-at-stg-in-trace '(MU RS1)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(defthm no-inst-at-MU-RS1-MT-step-if-not-dispatch-to-MU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (RS-valid? (MU-RS1 (MA-MU MA)))))
                  (or (not (b1p (dispatch-to-MU? MA)))
                      (not (b1p (select-MU-RS1? (MA-MU MA))))))
             (no-INST-at-stg '(MU RS1) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg)
                  :use (:instance no-inst-at-MU-RS1))))
)

(defthm no-INST-at-MU-RS1-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (RS-valid? (MU-RS1 (step-MU MA sigs))))))
             (no-inst-at-stg '(MU RS1) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-MU step-MU-RS1 lift-b-ops))))

;; Proof of uniq-INST-at-MU-lch1-MT-step
(defthm not-INST-stg-step-INST-MU-lch1-if-issue-MU-RS0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(MU RS0)))
                  (b1p (issue-MU-RS0? (MA-MU MA) MA)))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(MU lch1))))
  :hints (("goal" :use (:instance INST-is-at-one-of-the-stages)
                  :in-theory (e/d (step-inst-execute-inst
```

```
                                   step-INST-low-level-functions
                                   mu-stg-p lift-b-ops
                                   issue-mu-rs0? issue-mu-rs1?
                                   MU-RS0-ISSUE-READY?
                                   MU-RS1-ISSUE-READY?)
                                (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-MU-lch1-if-issue-MU-RS0-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(MU RS0) trace)
                  (b1p (issue-MU-RS0? (MA-MU MA) MA)))
             (no-inst-at-stg-in-trace '(MU lch1)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
(defthm uniq-inst-at-MU-lch1-if-issue-MU-RS0-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg-in-trace '(MU RS0) trace)
                  (b1p (issue-MU-RS0? (MA-MU MA) MA))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stg-in-trace '(MU lch1)
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc)))))

(defthm uniq-inst-at-MU-lch1-if-issue-MU-RS0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (issue-MU-RS0? (MA-MU MA) MA))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stg '(MU lch1) (MT-step MT MA sigs)))
  :hints (("goal" :use (:instance UNIQ-INST-AT-MU-RS0-IF-VALID)
                  :in-theory (enable uniq-inst-at-stg issue-MU-RS0?
                                     lift-b-ops MU-RS0-ISSUE-READY?))))
)

(defthm not-INST-stg-step-INST-MU-lch1-if-issue-MU-RS1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(MU RS1)))
                  (b1p (issue-MU-RS1? (MA-MU MA) MA)))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(MU lch1))))
  :hints (("goal" :use (:instance INST-is-at-one-of-the-stages)
                  :in-theory (e/d (step-inst-execute-inst
                                   step-INST-low-level-functions
                                   mu-stg-p lift-b-ops
                                   issue-mu-rs0? issue-mu-rs1?
                                   MU-RS0-ISSUE-READY?
                                   MU-RS1-ISSUE-READY?)
                                (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-MU-lch1-if-issue-MU-RS1-help-help
```

```
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(MU RS1) trace)
                  (b1p (issue-MU-RS1? (MA-MU MA) MA)))
             (no-inst-at-stg-in-trace '(MU lch1)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
(defthm uniq-inst-at-MU-lch1-if-issue-MU-RS1-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg-in-trace '(MU RS1) trace)
                  (b1p (issue-MU-RS1? (MA-MU MA) MA))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stg-in-trace '(MU lch1)
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc)))))

(defthm uniq-inst-at-MU-lch1-if-issue-MU-RS1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (issue-MU-RS1? (MA-MU MA) MA))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stg '(MU lch1) (MT-step MT MA sigs)))
  :hints (("goal" :use (:instance UNIQ-INST-AT-MU-RS1-IF-VALID)
                  :in-theory (enable uniq-inst-at-stg issue-MU-RS1?
                                     lift-b-ops MU-RS1-ISSUE-READY?))))
)

(defthm not-INST-stg-step-INST-MU-lch1-if-MU-lch1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(MU lch1)))
                  (b1p (MU-latch1-valid? (MU-lch1 (MA-MU MA))))
                  (b1p (MU-latch2-valid? (MU-lch2 (MA-MU MA))))
                  (not (b1p (CDB-for-MU? MA))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(MU lch1))))
  :hints (("goal" :use (:instance INST-is-at-one-of-the-stages)
                  :in-theory (e/d (step-inst-execute-inst
                                   step-inst-low-level-functions
                                   ISSUE-MU-RS1?
                                   issue-mu-rs0? lift-b-ops)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-MU-lch1-if-MU-lch1-valid-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(MU lch1) trace)
                  (b1p (MU-latch1-valid? (MU-lch1 (MA-MU MA))))
                  (b1p (MU-latch2-valid? (MU-lch2 (MA-MU MA))))
                  (not (b1p (CDB-for-MU? MA))))
             (no-inst-at-stg-in-trace '(MU lch1)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))
```

```
(local
(defthm uniq-inst-at-MU-lch1-if-MU-lch1-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg-in-trace '(MU lch1) trace)
                  (b1p (MU-latch1-valid? (MU-lch1 (MA-MU MA))))
                  (b1p (MU-latch2-valid? (MU-lch2 (MA-MU MA))))
                  (not (b1p (CDB-for-MU? MA)))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stg-in-trace '(MU lch1)
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc)))
  :hints (("goal" :in-theory (enable lift-b-ops)))))

(defthm uniq-inst-at-MU-lch1-if-MU-lch1-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (MU-latch1-valid? (MU-lch1 (MA-MU MA))))
                  (b1p (MU-latch2-valid? (MU-lch2 (MA-MU MA))))
                  (not (b1p (CDB-for-MU? MA)))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stg '(MU lch1) (MT-step MT MA sigs)))
  :hints (("goal" :use (:instance UNIQ-INST-AT-MU-LCH1-IF-VALID)
                  :in-theory (enable uniq-inst-at-stg))))
)


(defthm uniq-INST-at-MU-lch1-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (MU-latch1-valid? (MU-lch1 (step-MU MA sigs)))))
             (uniq-inst-at-stg '(MU lch1) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-MU step-MU-lch1 lift-b-ops))))

;; Proof of no-INST-at-MU-lch1-MT-step
(defthm not-INST-stg-step-INST-MU-lch1-if-not-issue
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (or (b1p (CDB-for-MU? MA))
                      (not (b1p (MU-latch2-valid? (MU-lch2 (MA-MU MA))))))
                  (not (b1p (issue-MU-RS0? (MA-MU MA) MA)))
                  (not (b1p (issue-MU-RS1? (MA-MU MA) MA))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(MU lch1))))
  :Hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-INST-execute-INST
                                     step-inst-low-level-functions
                                     lift-b-ops)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-INST-at-MU-lch1-MT-step-if-no-issue-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (or (b1p (CDB-for-MU? MA))
                      (not (b1p (MU-latch2-valid? (MU-lch2 (MA-MU MA))))))
                  (not (b1p (issue-MU-RS0? (MA-MU MA) MA)))
```

```
                (not (b1p (issue-MU-RS1? (MA-MU MA) MA))))
            (no-INST-at-stg-in-trace '(MU lch1)
                                     (step-trace trace MT MA sigs
                                                 ISA spc smc)))))

(defthm no-INST-at-MU-lch1-MT-step-if-no-issue
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (or (b1p (CDB-for-MU? MA))
                      (not (b1p (MU-latch2-valid? (MU-lch2 (MA-MU MA))))))
                  (not (b1p (issue-MU-RS0? (MA-MU MA) MA)))
                  (not (b1p (issue-MU-RS1? (MA-MU MA) MA))))
             (no-INST-at-stg '(MU lch1) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-INST-at-stg))))
)

(defthm not-INST-stg-step-INST-MU-lch1-if-not-MU-lch1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(MU lch1)))
                  (not (b1p (issue-MU-RS0? (MA-MU MA) MA)))
                  (not (b1p (issue-MU-RS1? (MA-MU MA) MA))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(MU lch1))))
  :hints (("goal" :in-theory (e/d (step-INST-execute-INST
                                   step-inst-low-level-functions)
                                  (inst-is-at-one-of-the-stages))
                  :use (:instance inst-is-at-one-of-the-stages))))

(encapsulate nil
(local
(defthm no-INST-at-MU-lch1-MT-step-if-lch1-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(MU lch1) trace)
                  (not (b1p (issue-MU-RS0? (MA-MU MA) MA)))
                  (not (b1p (issue-MU-RS1? (MA-MU MA) MA))))
             (no-INST-at-stg-in-trace '(MU lch1)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(defthm no-INST-at-MU-lch1-MT-step-if-lch1-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (MU-latch1-valid? (MU-lch1 (MA-MU MA)))))
                  (not (b1p (issue-MU-RS0? (MA-MU MA) MA)))
                  (not (b1p (issue-MU-RS1? (MA-MU MA) MA))))
             (no-INST-at-stg '(MU lch1) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg)
                  :use (:instance NO-INST-AT-MU-LCH1))))
)

(defthm no-INST-at-MU-lch1-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (MU-latch1-valid? (MU-lch1 (step-MU MA sigs))))))
             (no-inst-at-stg '(MU lch1) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-MU step-MU-lch1 lift-b-ops))))

;; Proof of uniq-INST-at-MU-lch2-MT-step
```

```
(defthm not-INST-stg-step-INST-MU-lch2-if-not-MU-lch1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(MU lch1)))
                  (or (b1p (CDB-for-MU? MA))
                      (not (b1p (MU-latch2-valid? (MU-lch2 (MA-MU MA)))))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(MU lch2))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-inst-execute-inst lift-b-ops
                                     execute-stg-p MU-STG-P
                                     step-inst-low-level-functions)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-MU-lch2-MT-step-if-MU-lch1-valid-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(MU lch1) trace)
                  (or (b1p (CDB-for-MU? MA))
                      (not (b1p (MU-latch2-valid? (MU-lch2 (MA-MU MA)))))))
             (no-inst-at-stg-in-trace '(MU lch2)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
(defthm uniq-inst-at-MU-lch2-MT-step-if-MU-lch1-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg-in-trace '(MU lch1) trace)
                  (or (b1p (CDB-for-MU? MA))
                      (not (b1p (MU-latch2-valid? (MU-lch2 (MA-MU MA))))))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stg-in-trace '(MU lch2)
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc)))
  :hints (("goal" :in-theory (enable lift-b-ops)))))

(defthm uniq-inst-at-MU-lch2-MT-step-if-MU-lch1-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (or (b1p (CDB-for-MU? MA))
                      (not (b1p (MU-latch2-valid? (MU-lch2 (MA-MU MA))))))
                  (b1p (MU-latch1-valid? (MU-lch1 (MA-MU MA))))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stg '(MU lch2) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable uniq-inst-at-stg)
                  :use (:instance UNIQ-INST-AT-MU-LCH1-IF-VALID))))
)

(defthm not-INST-stg-step-INST-MU-lch2-if-not-MU-lch2
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(MU lch2)))
                  (b1p (MU-latch2-valid? (MU-lch2 (MA-MU MA))))
                  (not (b1p (CDB-for-MU? MA))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
```

```
                       '(MU lch2))))
  :hints (("goal" :use (:instance INST-is-at-one-of-the-stages)
                  :in-theory (e/d (step-INST-execute-inst
                                   step-INST-low-level-functions
                                   execute-stg-p MU-stg-p lift-b-ops)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-MU-lch2-MT-step-if-MU-lch2-valid-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (MU-latch2-valid? (MU-lch2 (MA-MU MA))))
                  (no-inst-at-stg-in-trace '(MU lch2) trace)
                  (not (b1p (CDB-for-MU? MA))))
             (no-inst-at-stg-in-trace '(MU lch2)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
(defthm uniq-inst-at-MU-lch2-MT-step-if-MU-lch2-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg-in-trace '(MU lch2) trace)
                  (b1p (MU-latch2-valid? (MU-lch2 (MA-MU MA))))
                  (not (b1p (CDB-for-MU? MA)))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stg-in-trace '(MU lch2)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(defthm uniq-inst-at-MU-lch2-MT-step-if-MU-lch2-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (MU-latch2-valid? (MU-lch2 (MA-MU MA))))
                  (not (b1p (CDB-for-MU? MA)))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stg '(MU lch2) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable uniq-inst-at-stg)
                  :use (:instance UNIQ-INST-AT-MU-LCH2-IF-VALID))))
)

(defthm uniq-INST-at-MU-lch2-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (MU-latch2-valid? (MU-lch2 (step-MU MA sigs)))))
             (uniq-inst-at-stg '(MU lch2) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-MU step-MU-lch2 lift-b-ops))))

;; Proof of no-INST-at-MU-lch2-MT-step
(encapsulate nil
(local
(defthm no-INST-at-MU-lch2-MT-step-if-not-MU-lch1-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(MU lch1) trace)
                  (or (b1p (CDB-for-MU? MA))
                      (not (b1p (MU-latch2-valid? (MU-lch2 (MA-MU MA)))))))
             (no-inst-at-stg-in-trace '(MU lch2)
```

```
                                    (step-trace trace MT MA sigs
                                                ISA spc smc)))))

(defthm no-INST-at-MU-lch2-MT-step-if-not-MU-lch1-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (MU-latch1-valid? (MU-lch1 (MA-MU MA)))))
                  (or (b1p (CDB-for-MU? MA))
                      (not (b1p (MU-latch2-valid? (MU-lch2 (MA-MU MA)))))))
             (no-inst-at-stg '(MU lch2) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg)
                  :use (:instance no-inst-at-MU-lch1))))
)

(defthm no-INST-at-MU-lch2-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (MU-latch2-valid? (MU-lch2 (step-MU MA sigs))))))
             (no-inst-at-stg '(MU lch2) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-MU step-MU-lch2 lift-b-ops))))

(defthm no-MU-stg-conflict-preserved
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA))
             (no-MU-stg-conflict (MT-step MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable no-MU-stg-conflict))))

;;; Proof of no-LSU-stg-conflict-preserved
;; Proof of uniq-inst-at-LSU-RS0-MT-step
(defthm not-INST-stg-step-inst-LSU-RS0-if-not-LSU-RS0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(LSU RS0)))
                  (b1p (LSU-RS-valid? (LSU-RS0 (MA-LSU MA))))
                  (not (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU RS0))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-inst-execute-inst lift-b-ops
                                   step-inst-dq-inst dq-stg-p
                                   step-inst-low-level-functions
                                   DISPATCH-INST? LSU-READY?
                                   dispatch-to-LSU? SELECT-LSU-RS0?)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-LSU-RS0-MT-step-if-LSU-RS0-valid-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(LSU RS0) trace)
                  (b1p (LSU-RS-valid? (LSU-RS0 (MA-LSU MA))))
                  (not (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))))
             (no-inst-at-stg-in-trace '(LSU RS0)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))
  :hints (("goal" :in-theory (enable step-LSU step-LSU-RS0 lift-b-ops)))))

(local
(defthm uniq-inst-at-LSU-RS0-MT-step-if-LSU-RS0-valid-help
```

```
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg-in-trace '(LSU RS0) trace)
                  (b1p (LSU-RS-valid? (LSU-RS0 (MA-LSU MA))))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))))
             (uniq-inst-at-stg-in-trace '(LSU RS0)
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc)))
  :hints (("goal" :in-theory (enable step-LSU step-LSU-RS0 lift-b-ops)))))

(defthm uniq-inst-at-LSU-RS0-MT-step-if-LSU-RS0-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (LSU-RS-valid? (LSU-RS0 (MA-LSU MA))))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))))
             (uniq-inst-at-stg '(LSU RS0) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable uniq-inst-at-stg)
                  :use (:instance uniq-inst-at-LSU-RS0-if-valid))))
)


(defthm not-INST-stg-step-INST-LSU-RS0-if-not-DE0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(DQ 0)))
                  (not (b1p (LSU-RS-valid? (LSU-RS0 (MA-LSU MA)))))
                  (b1p (dispatch-to-LSU? MA))
                  (b1p (select-LSU-RS0? (MA-LSU MA))))
             (not (equal (INST-stg (step-inst i MT MA sigs))
                         '(LSU RS0))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-inst-execute-inst
                                     execute-stg-p step-inst-dq-inst
                                     dq-stg-p
                                     step-inst-low-level-functions)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-LSU-RS0-MT-step-if-dispatch-to-LSU-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(DQ 0) trace)
                  (not (b1p (LSU-RS-valid? (LSU-RS0 (MA-LSU MA)))))
                  (b1p (dispatch-to-LSU? MA))
                  (b1p (select-LSU-RS0? (MA-LSU MA))))
             (no-inst-at-stg-in-trace '(LSU RS0)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
(defthm uniq-inst-at-LSU-RS0-MT-step-if-dispatch-to-LSU-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg-in-trace '(DQ 0) trace)
                  (not (b1p (LSU-RS-valid? (LSU-RS0 (MA-LSU MA))))))
```

```
                (not (b1p (flush-all? MA sigs)))
                (b1p (dispatch-to-LSU? MA))
                (b1p (select-LSU-RS0? (MA-LSU MA))))
           (uniq-inst-at-stg-in-trace '(LSU RS0)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(defthm uniq-inst-at-LSU-RS0-MT-step-if-dispatch-to-LSU
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (LSU-RS-valid? (LSU-RS0 (MA-LSU MA)))))
                (not (b1p (flush-all? MA sigs)))
                (b1p (dispatch-to-LSU? MA))
                (b1p (select-LSU-RS0? (MA-LSU MA))))
           (uniq-inst-at-stg '(LSU RS0) (MT-step MT MA sigs)))
  :hints (("goal" :use (:instance UNIQ-INST-AT-STG-IF-DQ-DE0-VALID)
                :in-theory (enable dispatch-to-LSU? lift-b-ops
                                   uniq-inst-at-stg DQ-ready-to-LSU?))))
)

(defthm uniq-inst-at-LSU-RS0-MT-step
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (LSU-RS-valid? (LSU-RS0 (step-LSU MA sigs)))))
           (uniq-inst-at-stg '(LSU RS0) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-LSU step-LSU-RS0 lift-b-ops))))

;; Proof of no-inst-at-LSU-RS0-MT-step
(defthm not-INST-stg-step-INST-LSU-RS0-if-issue-LSU-RS0
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (LSU-RS-valid? (LSU-RS0 (MA-LSU MA))))
                (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs)))
           (not (equal (INST-stg (step-INST i MT MA sigs))
                       '(LSU RS0))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                :in-theory (e/d (step-inst-execute-inst
                                   step-INST-dq-inst dispatch-inst?
                                   dispatch-to-LSU? LSU-READY?
                                   lift-b-ops SELECT-LSU-RS0?
                                   step-inst-low-level-functions)
                                (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-LSU-RS0-MT-step-if-issue-LSU-RS0-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (LSU-RS-valid? (LSU-RS0 (MA-LSU MA))))
                (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs)))
           (no-inst-at-stg-in-trace '(LSU RS0)
                                    (step-trace trace MT MA sigs
                                                ISA spc smc)))))

(defthm no-inst-at-LSU-RS0-MT-step-if-issue-LSU-RS0
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (LSU-RS-valid? (LSU-RS0 (MA-LSU MA))))
                (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs)))
           (no-inst-at-stg '(LSU RS0) (MT-step MT MA sigs)))
```

```
  :hints (("goal" :in-theory (enable no-inst-at-stg))))
)

(defthm not-INST-stg-step-INST-LSU-RS0-if-not-dispatch
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(LSU RS0)))
                  (or (not (b1p (select-LSU-RS0? (MA-LSU MA))))
                      (not (b1p (dispatch-to-LSU? MA)))))
             (not (equal (INST-stg (step-inst i MT MA sigs))
                         '(LSU RS0))))
  :hints (("goal" :in-theory (e/d (step-INST-execute-inst
                                   step-INST-dq-inst
                                   step-INST-low-level-functions
                                   DISPATCH-INST? lift-b-ops
                                   DQ-stg-p)
                                  (inst-is-at-one-of-the-stages))
                  :use (:instance inst-is-at-one-of-the-stages))))

(encapsulate nil
(local
(defthm no-inst-at-LSU-RS0-MT-step-if-not-LSU-RS-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(LSU RS0) trace)
                  (or (not (b1p (select-LSU-RS0? (MA-LSU MA))))
                      (not (b1p (dispatch-to-LSU? MA)))))
             (no-inst-at-stg-in-trace '(LSU RS0)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(defthm no-inst-at-LSU-RS0-MT-step-if-not-LSU-RS-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (LSU-RS-valid? (LSU-RS0 (MA-LSU MA)))))
                  (or (not (b1p (select-LSU-RS0? (MA-LSU MA))))
                      (not (b1p (dispatch-to-LSU? MA)))))
             (no-inst-at-stg '(LSU RS0) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg)
                  :use (:instance no-inst-at-LSU-RS0))))
)

(defthm no-inst-at-LSU-RS0-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (LSU-RS-valid? (LSU-RS0 (step-LSU MA sigs))))))
             (no-inst-at-stg '(LSU RS0) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-LSU step-LSU-RS0 lift-b-ops))))

;; Proofs of uniq-inst-at-LSU-RS1-MT-step
(defthm not-INST-stg-step-inst-LSU-RS1-if-not-LSU-RS1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(LSU RS1)))
                  (b1p (LSU-RS-valid? (LSU-RS1 (MA-LSU MA))))
                  (not (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU RS1))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
```

```
                  :in-theory (e/d (step-inst-execute-inst lift-b-ops
                                   step-inst-dq-inst dq-stg-p
                                   step-inst-low-level-functions
                                   DISPATCH-INST? LSU-READY?
                                   dispatch-to-LSU? SELECT-LSU-RS1?
                                   SELECT-LSU-RS0?)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-LSU-RS1-MT-step-if-LSU-RS1-valid-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(LSU RS1) trace)
                  (b1p (LSU-RS-valid? (LSU-RS1 (MA-LSU MA))))
                  (not (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))))
             (no-inst-at-stg-in-trace '(LSU RS1)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))
  :hints (("goal" :in-theory (enable step-LSU step-LSU-RS1 lift-b-ops)))))

(local
(defthm uniq-inst-at-LSU-RS1-MT-step-if-LSU-RS1-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg-in-trace '(LSU RS1) trace)
                  (b1p (LSU-RS-valid? (LSU-RS1 (MA-LSU MA))))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))))
             (uniq-inst-at-stg-in-trace '(LSU RS1)
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc)))
  :hints (("goal" :in-theory (enable step-LSU step-LSU-RS1 lift-b-ops)))))

(defthm uniq-inst-at-LSU-RS1-MT-step-if-LSU-RS1-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (LSU-RS-valid? (LSU-RS1 (MA-LSU MA))))
                  (not (b1p (flush-all? MA sigs)))
                  (not (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))))
             (uniq-inst-at-stg '(LSU RS1) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable uniq-inst-at-stg)
                  :use (:instance uniq-inst-at-LSU-RS1-if-valid))))
)

(defthm not-INST-stg-step-INST-LSU-RS1-if-not-DE0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(DQ 0)))
                  (not (b1p (LSU-RS-valid? (LSU-RS1 (MA-LSU MA)))))
                  (b1p (dispatch-to-LSU? MA))
                  (b1p (select-LSU-RS1? (MA-LSU MA))))
             (not (equal (INST-stg (step-inst i MT MA sigs))
                         '(LSU RS1))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-inst-execute-inst
                                   execute-stg-p step-inst-dq-inst
                                   dq-stg-p
                                   step-inst-low-level-functions)
```

```
                                (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-LSU-RS1-MT-step-if-dispatch-to-LSU-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(DQ 0) trace)
                  (not (b1p (LSU-RS-valid? (LSU-RS1 (MA-LSU MA)))))
                  (b1p (dispatch-to-LSU? MA))
                  (b1p (select-LSU-RS1? (MA-LSU MA))))
             (no-inst-at-stg-in-trace '(LSU RS1)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
(defthm uniq-inst-at-LSU-RS1-MT-step-if-dispatch-to-LSU-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg-in-trace '(DQ 0) trace)
                  (not (b1p (LSU-RS-valid? (LSU-RS1 (MA-LSU MA)))))
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (dispatch-to-LSU? MA))
                  (b1p (select-LSU-RS1? (MA-LSU MA))))
             (uniq-inst-at-stg-in-trace '(LSU RS1)
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc)))
  :hints (("goal" :in-theory (enable SELECT-LSU-RS1? SELECT-LSU-RS0?
                                     lift-b-ops)))))

(defthm uniq-inst-at-LSU-RS1-MT-step-if-dispatch-to-LSU
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (LSU-RS-valid? (LSU-RS1 (MA-LSU MA)))))
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (dispatch-to-LSU? MA))
                  (b1p (select-LSU-RS1? (MA-LSU MA))))
             (uniq-inst-at-stg '(LSU RS1) (MT-step MT MA sigs)))
  :hints (("goal" :use (:instance UNIQ-INST-AT-STG-IF-DQ-DE0-VALID)
                  :in-theory (enable dispatch-to-LSU? lift-b-ops
                                     uniq-inst-at-stg DQ-ready-to-LSU?))))
)

(defthm uniq-inst-at-LSU-RS1-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (LSU-RS-valid? (LSU-RS1 (step-LSU MA sigs)))))
             (uniq-inst-at-stg '(LSU RS1) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-LSU step-LSU-RS1 lift-b-ops))))

;; Proofs of no-inst-at-LSU-RS1-MT-step
(defthm not-INST-stg-step-INST-LSU-RS1-if-issue-LSU-RS1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (LSU-RS-valid? (LSU-RS1 (MA-LSU MA))))
                  (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs)))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU RS1))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
```

```
                :in-theory (e/d (step-inst-execute-inst
                                 step-INST-dq-inst dispatch-inst?
                                 dispatch-to-LSU? LSU-READY?
                                 lift-b-ops SELECT-LSU-RS1?
                                 select-LSU-RS0?
                                 step-inst-low-level-functions)
                                (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-LSU-RS1-MT-step-if-issue-LSU-RS1-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (LSU-RS-valid? (LSU-RS1 (MA-LSU MA))))
                  (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs)))
             (no-inst-at-stg-in-trace '(LSU RS1)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(defthm no-inst-at-LSU-RS1-MT-step-if-issue-LSU-RS1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (LSU-RS-valid? (LSU-RS1 (MA-LSU MA))))
                  (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs)))
             (no-inst-at-stg '(LSU RS1) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg))))
)

(defthm not-INST-stg-step-INST-LSU-RS1-if-not-dispatch
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(LSU RS1)))
                  (or (not (b1p (select-LSU-RS1? (MA-LSU MA))))
                      (not (b1p (dispatch-to-LSU? MA)))))
             (not (equal (INST-stg (step-inst i MT MA sigs))
                         '(LSU RS1))))
  :hints (("goal" :in-theory (e/d (step-INST-execute-inst
                                   step-INST-dq-inst
                                   step-INST-low-level-functions
                                   DISPATCH-INST? lift-b-ops
                                   select-LSU-RS1? select-LSU-RS0?
                                   DQ-stg-p)
                                  (inst-is-at-one-of-the-stages))
                  :use (:instance inst-is-at-one-of-the-stages))))

(encapsulate nil
(local
(defthm no-inst-at-LSU-RS1-MT-step-if-not-LSU-RS-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(LSU RS1) trace)
                  (or (not (b1p (select-LSU-RS1? (MA-LSU MA))))
                      (not (b1p (dispatch-to-LSU? MA)))))
             (no-inst-at-stg-in-trace '(LSU RS1)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(defthm no-inst-at-LSU-RS1-MT-step-if-not-LSU-RS-valid
    (implies (and (inv MT MA)
```

```
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (LSU-RS-valid? (LSU-RS1 (MA-LSU MA)))))
                (or (not (b1p (select-LSU-RS1? (MA-LSU MA))))
                    (not (b1p (dispatch-to-LSU? MA)))))
           (no-inst-at-stg '(LSU RS1) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg)
                  :use (:instance no-inst-at-LSU-RS1))))
)

(defthm no-inst-at-LSU-RS1-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (LSU-RS-valid? (LSU-RS1 (step-LSU MA sigs))))))
             (no-inst-at-stg '(LSU RS1) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-LSU step-LSU-RS1 lift-b-ops))))

;; Proof of uniq-inst-at-LSU-rbuf-MT-step
(defthm not-INST-stg-step-inst-LSU-rbuf-if-not-LSU-rbuf
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(LSU rbuf)))
                  (b1p (rbuf-valid? (LSU-rbuf (MA-LSU MA))))
                  (not (b1p (release-rbuf? (MA-LSU MA) MA sigs))))
             (not (equal (INST-stg (step-inst i MT MA sigs))
                         '(LSU rbuf))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-inst-execute-inst
                                   ISSUE-LSU-RS0? lift-b-ops
                                   ISSUE-LSU-RS1?
                                   step-inst-low-level-functions)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-LSU-rbuf-MT-step-if-rbuf-valid-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(LSU rbuf) trace)
                  (b1p (rbuf-valid? (LSU-rbuf (MA-LSU MA))))
                  (not (b1p (release-rbuf? (MA-LSU MA) MA sigs)))
                  (not (b1p (flush-all? MA sigs))))
             (no-inst-at-stg-in-trace '(LSU rbuf)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
(defthm uniq-inst-at-LSU-rbuf-MT-step-if-rbuf-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg-in-trace '(LSU rbuf) trace)
                  (b1p (rbuf-valid? (LSU-rbuf (MA-LSU MA))))
                  (not (b1p (release-rbuf? (MA-LSU MA) MA sigs)))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stg-in-trace '(LSU rbuf)
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc)))))

(defthm uniq-inst-at-LSU-rbuf-MT-step-if-rbuf-valid
    (implies (and (inv MT MA)
```

```
                (MAETT-p MT) (MA-state-p MA)
                (b1p (rbuf-valid? (LSU-rbuf (MA-LSU MA))))
                (not (b1p (release-rbuf? (MA-LSU MA) MA sigs)))
                (not (b1p (flush-all? MA sigs))))
           (uniq-inst-at-stg '(LSU rbuf)
                             (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable uniq-inst-at-stg)
                  :use (:instance uniq-inst-at-LSU-rbuf-if-valid))))
)

(defthm not-INST-stg-step-INST-LSU-rbuf-if-not-LSU-RS0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(LSU RS0)))
                  (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                  (not (b1p (LSU-RS-ld-st? (LSU-RS0 (MA-LSU MA))))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU rbuf))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-inst-execute-inst
                                   step-inst-low-level-functions
                                   ISSUE-LSU-RS0? ISSUE-LSU-RS1?
                                   LSU-RS0-ISSUE-READY?
                                   LSU-RS1-ISSUE-READY?
                                   lift-b-ops)
                                  (inst-is-at-one-of-the-stages)))))
(encapsulate nil
(local
(defthm uniq-inst-at-LSU-rbuf-MT-step-if-issue-RS0-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(LSU RS0) trace)
                  (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                  (not (b1p (LSU-RS-ld-st? (LSU-RS0 (MA-LSU MA)))))
                  (not (b1p (flush-all? MA sigs))))
             (no-inst-at-stg-in-trace '(LSU rbuf)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
(defthm uniq-inst-at-LSU-rbuf-MT-step-if-issue-RS0-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg-in-trace '(LSU RS0) trace)
                  (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                  (not (b1p (LSU-RS-ld-st? (LSU-RS0 (MA-LSU MA)))))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stg-in-trace '(LSU rbuf)
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc)))))

(defthm uniq-inst-at-LSU-rbuf-MT-step-if-issue-RS0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                  (not (b1p (LSU-RS-ld-st? (LSU-RS0 (MA-LSU MA)))))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stg '(LSU rbuf) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable uniq-inst-at-stg lift-b-ops
```

```
                                    issue-LSU-RS0? LSU-RS0-ISSUE-READY?)
                  :use (:instance uniq-inst-at-LSU-RS0-if-valid))))
)
(defthm not-inst-stg-step-inst-LSU-rbuf-if-not-LSU-rs1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(LSU RS1)))
                  (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                  (not (b1p (LSU-RS-ld-st? (LSU-RS1 (MA-LSU MA)))))
                  (not (b1p (flush-all? MA sigs))))
             (not (equal (INST-stg (step-inst i MT MA sigs))
                         '(LSU rbuf))))
  :hints (("goal" :in-theory (e/d (step-inst-execute-inst
                                    step-inst-low-level-functions
                                    ISSUE-LSU-RS0? ISSUE-LSU-RS1?
                                    LSU-RS0-ISSUE-READY?
                                    LSU-RS1-ISSUE-READY?
                                    lift-b-ops)
                                  (inst-is-at-one-of-the-stages))
                  :use (:instance inst-is-at-one-of-the-stages))))

(encapsulate nil
(local
(defthm uniq-inst-at-LSU-rbuf-MT-step-if-issue-RS1-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(LSU RS1) trace)
                  (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                  (not (b1p (LSU-RS-ld-st? (LSU-RS1 (MA-LSU MA)))))
                  (not (b1p (flush-all? MA sigs))))
             (no-inst-at-stg-in-trace '(LSU rbuf)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(local
(defthm uniq-inst-at-LSU-rbuf-MT-step-if-issue-RS1-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg-in-trace '(LSU RS1) trace)
                  (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                  (not (b1p (LSU-RS-ld-st? (LSU-RS1 (MA-LSU MA)))))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stg-in-trace '(LSU rbuf)
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc)))))

(defthm uniq-inst-at-LSU-rbuf-MT-step-if-issue-RS1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                  (not (b1p (LSU-RS-ld-st? (LSU-RS1 (MA-LSU MA)))))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stg '(LSU rbuf) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable uniq-inst-at-stg lift-b-ops
                                     issue-LSU-RS1? LSU-RS1-ISSUE-READY?)
                  :use (:instance uniq-inst-at-LSU-RS1-if-valid))))
)

(defthm uniq-inst-at-LSU-rbuf-MT-step
```

```
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (rbuf-valid? (LSU-rbuf (step-LSU MA sigs)))))
             (uniq-inst-at-stg '(LSU rbuf) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-LSU step-rbuf lift-b-ops))))

;; Proof of no-inst-at-LSU-rbuf-MT-step
(defthm not-INST-stg-step-INST-LSU-rbuf
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(LSU rbuf)))
                  (or (not (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs)))
                      (b1p (LSU-RS-ld-st? (LSU-RS0 (MA-LSU MA)))))
                  (or (not (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs)))
                      (b1p (LSU-RS-ld-st? (LSU-RS1 (MA-LSU MA))))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(LSU rbuf))))
  :hints (("goal" :in-theory (e/d (step-inst-execute-inst
                                   lift-b-ops
                                   step-inst-low-level-functions)
                                  (inst-is-at-one-of-the-stages))
                  :use (:instance inst-is-at-one-of-the-stages))))


(encapsulate nil
(local
(defthm no-inst-at-LSU-rbuf-MT-step-if-rbuf-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(LSU rbuf) trace)
                  (or (not (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs)))
                      (b1p (LSU-RS-ld-st? (LSU-RS0 (MA-LSU MA)))))
                  (or (not (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs)))
                      (b1p (LSU-RS-ld-st? (LSU-RS1 (MA-LSU MA))))))
             (no-inst-at-stg-in-trace '(LSU rbuf)
                          (step-trace trace MT MA sigs ISA spc smc)))))

(defthm no-inst-at-LSU-rbuf-MT-step-if-rbuf-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (rbuf-valid? (LSU-rbuf (MA-LSU MA)))))
                  (or (not (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs)))
                      (b1p (LSU-RS-ld-st? (LSU-RS0 (MA-LSU MA)))))
                  (or (not (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs)))
                      (b1p (LSU-RS-ld-st? (LSU-RS1 (MA-LSU MA))))))
             (no-inst-at-stg '(LSU rbuf) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg)
                  :use (:instance no-inst-at-LSU-rbuf))))
)

(defthm not-inst-stg-step-inst-LSU-rbuf-if-release-rbuf
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (release-rbuf? (MA-LSU MA) MA sigs))
                  (or (not (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs)))
                      (b1p (LSU-RS-ld-st? (LSU-RS0 (MA-LSU MA)))))
                  (or (not (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs)))
                      (b1p (LSU-RS-ld-st? (LSU-RS1 (MA-LSU MA))))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
```

```
                        '(LSU rbuf))))
  :hints (("goal" :in-theory (e/d (step-inst-execute-inst
                                   step-inst-low-level-functions)
                                  (inst-is-at-one-of-the-stages))
                  :use (:instance inst-is-at-one-of-the-stages))))

(encapsulate nil
(local
(defthm no-inst-at-LSU-rbuf-MT-step-if-release-rbuf-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (release-rbuf? (MA-LSU MA) MA sigs))
                  (or (not (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs)))
                      (b1p (LSU-RS-ld-st? (LSU-RS0 (MA-LSU MA)))))
                  (or (not (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs)))
                      (b1p (LSU-RS-ld-st? (LSU-RS1 (MA-LSU MA))))))
             (no-inst-at-stg-in-trace '(LSU rbuf)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))
  :hints (("goal" :in-theory (enable step-LSU step-rbuf lift-b-ops)))))

(defthm no-inst-at-LSU-rbuf-MT-step-if-release-rbuf
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (release-rbuf? (MA-LSU MA) MA sigs))
                  (or (not (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs)))
                      (b1p (LSU-RS-ld-st? (LSU-RS0 (MA-LSU MA)))))
                  (or (not (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs)))
                      (b1p (LSU-RS-ld-st? (LSU-RS1 (MA-LSU MA))))))
             (no-inst-at-stg '(LSU rbuf) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg))))
)

(defthm no-inst-at-LSU-rbuf-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (rbuf-valid? (LSU-rbuf (step-LSU MA sigs))))))
             (no-inst-at-stg '(LSU rbuf) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-LSU step-rbuf lift-b-ops))))

;; Proof of uniq-inst-at-LSU-wbuf0-MT-step
(encapsulate nil
(local
(defthm not-wbuf0-step-inst-if-not-LSU-RS0-help
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (execute-stg-p (INST-stg i))
                  (not (equal (INST-stg i) '(LSU RS0)))
                  (or (not (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
                      (and (b1p (release-wbuf0? (MA-LSU MA) sigs))
                           (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))))
                  (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                  (b1p (LSU-RS-LD-st? (LSU-RS0 (MA-LSU MA)))))
             (not (member-equal (INST-stg (step-INST i MT MA sigs))
                                '((LSU wbuf0)
                                  (LSU wbuf0 lch)
                                  (complete wbuf0) (commit wbuf0)))))
  :hints (("goal" :in-theory (enable step-inst-execute-inst
                                     EXECUTE-STG-P LSU-stg-p
                                     release-wbuf0? lift-b-ops
```

```
                                    RELEASE-WBUF0-READY?
                                    ISSUE-LSU-RS0? ISSUE-LSU-RS0?
                                    LSU-RS0-ISSUE-READY?
                                    LSU-RS1-ISSUE-READY?
                                    ISSUE-LSU-RS1?
                                    step-inst-low-level-functions)))))

(defthm not-wbuf0-step-inst-if-not-LSU-RS0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(LSU RS0)))
                  (or (not (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
                      (and (b1p (release-wbuf0? (MA-LSU MA) sigs))
                           (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))))
                  (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                  (b1p (LSU-RS-LD-st? (LSU-RS0 (MA-LSU MA)))))
             (not (member-equal (INST-stg (step-INST i MT MA sigs))
                                '((LSU wbuf0)
                                  (LSU wbuf0 lch)
                                  (complete wbuf0) (commit wbuf0)))))
  :hints (("goal" :in-theory (e/d (commit-stg-p lift-b-ops
                                   step-inst-commit-inst
                                   release-wbuf0?
                                   complete-stg-p
                                   RELEASE-WBUF0-READY?
                                   step-inst-complete-inst
                                   step-inst-low-level-functions)
                                  (inst-is-at-one-of-the-stages))
                  :use (:instance inst-is-at-one-of-the-stages))))
)

(encapsulate nil
(local
(defthm uniq-inst-at-LSU-wbuf0-MT-step-if-issue-RS0-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(LSU RS0) trace)
                  (or (not (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
                      (and (b1p (release-wbuf0? (MA-LSU MA) sigs))
                           (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))))
                  (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                  (b1p (LSU-RS-LD-st? (LSU-RS0 (MA-LSU MA)))))
             (no-inst-at-stgs-in-trace '((LSU wbuf0)
                                         (LSU wbuf0 lch)
                                         (complete wbuf0) (commit wbuf0))
                                       (step-trace trace MT MA sigs
                                                   ISA spc smc)))
  :hints (("goal" :in-theory (e/d (INST-SELECT-WBUF0? lift-b-ops)
                                  (member-equal))))))

(local
(defthm uniq-inst-at-LSU-wbuf0-MT-step-if-issue-RS0-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg-in-trace '(LSU RS0) trace)
                  (or (not (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
                      (and (b1p (release-wbuf0? (MA-LSU MA) sigs))
                           (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))))
                  (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
```

```
                (b1p (LSU-RS-LD-st? (LSU-RS0 (MA-LSU MA))))
                (not (b1p (flush-all? MA sigs))))
           (uniq-inst-at-stgs-in-trace '((LSU wbuf0)
                                         (LSU wbuf0 lch)
                                         (complete wbuf0) (commit wbuf0))
                                 (step-trace trace MT MA sigs
                                             ISA spc smc)))
  :hints (("goal" :in-theory (e/d (INST-SELECT-WBUF0? lift-b-ops)
                                  (member-equal))))))

(defthm uniq-inst-at-LSU-wbuf0-MT-step-if-issue-RS0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (or (not (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
                      (and (b1p (release-wbuf0? (MA-LSU MA) sigs))
                           (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))))
                  (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                  (b1p (LSU-RS-LD-st? (LSU-RS0 (MA-LSU MA))))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stgs '((LSU wbuf0) (LSU wbuf0 lch)
                                  (complete wbuf0) (commit wbuf0))
                                (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable uniq-inst-at-stgs uniq-inst-at-stg
                                     ISSUE-LSU-RS0? LSU-RS0-ISSUE-READY?
                                     lift-b-ops)
                  :use (:instance uniq-inst-at-LSU-RS0-if-valid))))
)

(encapsulate nil
(local
(defthm not-wbuf0-step-inst-if-not-LSU-RS1-help
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (execute-stg-p (INST-stg i))
                  (not (equal (INST-stg i) '(LSU RS1)))
                  (or (not (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
                      (and (b1p (release-wbuf0? (MA-LSU MA) sigs))
                           (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))))
                  (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                  (b1p (LSU-RS-LD-st? (LSU-RS1 (MA-LSU MA)))))
             (not (member-equal (INST-stg (step-INST i MT MA sigs))
                                '((LSU wbuf0)
                                  (LSU wbuf0 lch)
                                  (complete wbuf0) (commit wbuf0)))))
  :hints (("goal" :in-theory (enable step-inst-execute-inst
                                     EXECUTE-STG-P LSU-stg-p
                                     release-wbuf0? lift-b-ops
                                     RELEASE-WBUF0-READY?
                                     ISSUE-LSU-RS0? ISSUE-LSU-RS0?
                                     LSU-RS0-ISSUE-READY?
                                     LSU-RS1-ISSUE-READY?
                                     ISSUE-LSU-RS1?
                                     step-inst-low-level-functions)))))

(defthm not-wbuf0-step-inst-if-not-LSU-RS1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(LSU RS1)))
                  (or (not (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
                      (and (b1p (release-wbuf0? (MA-LSU MA) sigs))
```

```
                          (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))))
                (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                (b1p (LSU-RS-LD-st? (LSU-RS1 (MA-LSU MA)))))
           (not (member-equal (INST-stg (step-INST i MT MA sigs))
                              '((LSU wbuf0)
                                (LSU wbuf0 lch)
                                (complete wbuf0) (commit wbuf0)))))
  :hints (("goal" :in-theory (e/d (commit-stg-p lift-b-ops
                                   step-inst-commit-inst
                                   release-wbuf0?
                                   complete-stg-p
                                   RELEASE-WBUF0-READY?
                                   step-inst-complete-inst
                                   step-inst-low-level-functions)
                                  (inst-is-at-one-of-the-stages))
                :use (:instance inst-is-at-one-of-the-stages))))
)

(encapsulate nil
(local
(defthm uniq-inst-at-LSU-wbuf0-MT-step-if-issue-RS1-help-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (no-inst-at-stg-in-trace '(LSU RS1) trace)
                (or (not (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
                    (and (b1p (release-wbuf0? (MA-LSU MA) sigs))
                         (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))))
                (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                (b1p (LSU-RS-LD-st? (LSU-RS1 (MA-LSU MA)))))
           (no-inst-at-stgs-in-trace '((LSU wbuf0)
                                       (LSU wbuf0 lch)
                                       (complete wbuf0) (commit wbuf0))
                                     (step-trace trace MT MA sigs
                                                 ISA spc smc)))
  :hints (("goal" :in-theory (e/d (INST-SELECT-WBUF0? lift-b-ops)
                                  (member-equal))))))

(local
(defthm uniq-inst-at-LSU-wbuf0-MT-step-if-issue-RS1-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (uniq-inst-at-stg-in-trace '(LSU RS1) trace)
                (or (not (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
                    (and (b1p (release-wbuf0? (MA-LSU MA) sigs))
                         (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))))
                (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                (b1p (LSU-RS-LD-st? (LSU-RS1 (MA-LSU MA))))
                (not (b1p (flush-all? MA sigs))))
           (uniq-inst-at-stgs-in-trace '((LSU wbuf0)
                                         (LSU wbuf0 lch)
                                         (complete wbuf0) (commit wbuf0))
                                       (step-trace trace MT MA sigs
                                                   ISA spc smc)))
  :hints (("goal" :in-theory (e/d (INST-SELECT-WBUF0? lift-b-ops)
                                  (member-equal))))))

(defthm uniq-inst-at-LSU-wbuf0-MT-step-if-issue-RS1
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (or (not (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
```

```
                    (and (b1p (release-wbuf0? (MA-LSU MA) sigs))
                         (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))))
               (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
               (b1p (LSU-RS-LD-st? (LSU-RS1 (MA-LSU MA))))
               (not (b1p (flush-all? MA sigs))))
          (uniq-inst-at-stgs '((LSU wbuf0) (LSU wbuf0 lch)
                               (complete wbuf0) (commit wbuf0))
                             (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable uniq-inst-at-stgs uniq-inst-at-stg
                                     ISSUE-LSU-RS1? LSU-RS1-ISSUE-READY?
                                     lift-b-ops)
               :use (:instance uniq-inst-at-LSU-RS1-if-valid))))
)

(defthm member-equal-step-INST-wbuf0-if-release-wbuf0
    (implies (and (inv MT MA)
               (INST-in i MT) (INST-p i)
               (MAETT-p MT) (MA-state-p MA)
               (not (b1p (flush-all? MA sigs)))
               (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
               (b1p (release-wbuf0? (MA-LSU MA) sigs)))
          (iff (member-equal (INST-stg (step-INST i MT MA sigs))
                             '((LSU wbuf0)
                               (LSU wbuf0 lch)
                               (complete wbuf0) (commit wbuf0)))
               (member-equal (INST-stg i) '((LSU wbuf1)
                                            (LSU wbuf1 lch)
                                            (complete wbuf1)
                                            (commit wbuf1)))))
  :hints (("goal" :in-theory (e/d (lift-b-ops release-wbuf0?
                                     check-wbuf1? flush-all?
                                     commit-stg-p
                                     step-inst-commit-inst
                                     step-inst-complete-inst
                                     complete-stg-p
                                     execute-stg-p LSU-STG-P
                                     step-inst-execute-inst
                                     INST-SELECT-WBUF0?
                                     RELEASE-WBUF0-READY?
                                     step-inst-low-level-functions)
                                   (inst-is-at-one-of-the-stages))
               :use (:instance inst-is-at-one-of-the-stages))))

(encapsulate nil
(local
(defthm uniq-inst-at-LSU-wbuf0-MT-step-if-wbuf1-valid-help-help
    (implies (and (inv MT MA)
               (subtrace-p trace MT) (INST-listp trace)
               (MAETT-p MT) (MA-state-p MA)
               (not (b1p (flush-all? MA sigs)))
               (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
               (no-inst-at-stgs-in-trace '((LSU wbuf1)
                                           (LSU wbuf1 lch)
                                           (complete wbuf1)
                                           (commit wbuf1))
                                         trace)
               (b1p (release-wbuf0? (MA-LSU MA) sigs)))
          (no-inst-at-stgs-in-trace '((LSU wbuf0)
                                      (LSU wbuf0 lch)
                                      (complete wbuf0) (commit wbuf0))
                                    (step-trace trace MT MA sigs
                                                ISA spc smc)))))
```

```
(local
(defthm uniq-inst-at-LSU-wbuf0-MT-step-if-wbuf1-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stgs-in-trace '((LSU wbuf1)
                                                (LSU wbuf1 lch)
                                                (complete wbuf1)
                                                (commit wbuf1))
                                              trace)
                  (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                  (b1p (release-wbuf0? (MA-LSU MA) sigs))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stgs-in-trace '((LSU wbuf0)
                                           (LSU wbuf0 lch)
                                           (complete wbuf0) (commit wbuf0))
                                         (step-trace trace MT MA sigs
                                                     ISA spc smc)))
  :hints (("goal" :in-theory (e/d () (member-equal))))))

(defthm uniq-inst-at-LSU-wbuf0-MT-step-if-wbuf1-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                  (b1p (release-wbuf0? (MA-LSU MA) sigs))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stgs '((LSU wbuf0) (LSU wbuf0 lch)
                                  (complete wbuf0) (commit wbuf0))
                                (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable uniq-inst-at-stgs)
                  :use (:instance UNIQ-INST-AT-LSU-WBUF1-IF-VALID))))
)

(defthm not-step-INST-at-wbuf0-if-not-commit-wbuf1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(commit wbuf1)))
                  (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                  (b1p (release-wbuf0? (MA-LSU MA) sigs))
                  (b1p (wbuf-commit? (LSU-wbuf1 (MA-LSU MA)))))
             (not (member-equal (INST-stg (step-INST i MT MA sigs))
                                '((LSU wbuf0)
                                  (LSU wbuf0 lch)
                                  (complete wbuf0) (commit wbuf0)))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-inst-low-level-functions
                                     step-inst-commit-inst
                                     step-inst-execute-inst
                                     step-inst-complete-inst
                                     INST-SELECT-WBUF0? lift-b-ops
                                     RELEASE-WBUF0?
                                     RELEASE-WBUF0-READY?
                                     execute-stg-p LSU-stg-p
                                     complete-stg-p
                                     commit-stg-p)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm not-uniq-inst-at-commit-wbuf0-help
```

```
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (subtrace-p trace MT) (INST-listp trace)
                  (subtrace-after-p i trace MT)
                  (not (committed-p i))
                  (MAETT-p MT) (MA-state-p MA))
             (not (uniq-inst-at-stg-in-trace '(commit wbuf0) trace)))
  :hints ((when-found-multiple ((EQUAL (INST-STG (CAR TRACE)) '(COMMIT WBUF0))
                                (COMMITTED-P I)
                                (SUBTRACE-AFTER-P I TRACE MT))
              (:use  (:instance INST-IN-ORDER-COMMIT-UNCOMMIT
                                (i (car trace)) (j i))))
          ("goal" :in-theory (disable INST-IN-ORDER-COMMIT-UNCOMMIT)))))

(defthm not-uniq-inst-at-commit-wbuf0-in-trace-cdr-if-car-is-not-commit
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (not (committed-p (car trace)))
                  (MAETT-p MT) (MA-state-p MA))
             (not (uniq-inst-at-stg-in-trace '(commit wbuf0) (cdr trace))))
  :hints (("goal" :use (:instance not-uniq-inst-at-commit-wbuf0-help
                                  (i (car trace)) (trace (cdr trace))))))
)

(encapsulate nil
(local
(defthm not-uniq-inst-at-commit-wbuf1-help
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (subtrace-p trace MT) (INST-listp trace)
                  (subtrace-after-p i trace MT)
                  (not (committed-p i))
                  (MAETT-p MT) (MA-state-p MA))
             (not (uniq-inst-at-stg-in-trace '(commit wbuf1) trace)))
  :hints ((when-found-multiple ((EQUAL (INST-STG (CAR TRACE)) '(COMMIT WBUF1))
                                (COMMITTED-P I)
                                (SUBTRACE-AFTER-P I TRACE MT))
              (:use  (:instance INST-IN-ORDER-COMMIT-UNCOMMIT
                                (i (car trace)) (j i))))
          ("goal" :in-theory (disable INST-IN-ORDER-COMMIT-UNCOMMIT)))))

(defthm not-uniq-inst-at-commit-wbuf1-in-trace-cdr-if-car-is-not-commit
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (not (committed-p (car trace)))
                  (MAETT-p MT) (MA-state-p MA))
             (not (uniq-inst-at-stg-in-trace '(commit wbuf1) (cdr trace))))
  :hints (("goal" :use (:instance not-uniq-inst-at-commit-wbuf1-help
                                  (i (car trace)) (trace (cdr trace))))))
)

(defthm not-inst-exint-now-car-if-not-uniq-inst-at-commit-wbuf10
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg-in-trace '(commit wbuf0) (cdr trace)))
             (equal (INST-exintr-now? (car trace) MA sigs) 0))
  :hints (("goal" :in-theory (enable INST-exintr-now?))))

(defthm not-inst-cause-jmp-car-if-not-uniq-inst-at-commit-wbuf0
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
```

```
                (MAETT-p MT) (MA-state-p MA)
                (uniq-inst-at-stg-in-trace '(commit wbuf0) (cdr trace)))
           (equal (INST-cause-jmp? (car trace) MT MA sigs) 0))
  :hints (("goal" :in-theory (enable INST-cause-jmp?))))

(defthm not-inst-exint-now-car-if-not-uniq-inst-at-commit-wbuf1
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (uniq-inst-at-stg-in-trace '(commit wbuf1) (cdr trace)))
           (equal  (INST-exintr-now? (car trace) MA sigs) 0))
  :hints (("goal" :in-theory (enable INST-exintr-now?))))

(defthm not-inst-cause-jmp-car-if-not-uniq-inst-at-commit-wbuf1
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (uniq-inst-at-stg-in-trace '(commit wbuf1) (cdr trace)))
           (equal (INST-cause-jmp? (car trace) MT MA sigs) 0))
  :hints (("goal" :in-theory (enable INST-cause-jmp?))))

(encapsulate nil
(local
(defthm uniq-inst-at-LSU-wbuf0-MT-step-if-wbuf1-commit-help-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (no-inst-at-stg-in-trace '(commit wbuf1) trace)
                (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                (b1p (release-wbuf0? (MA-LSU MA) sigs))
                (b1p (wbuf-commit? (LSU-wbuf1 (MA-LSU MA)))))
           (no-inst-at-stgs-in-trace '((LSU wbuf0)
                                       (LSU wbuf0 lch)
                                       (complete wbuf0) (commit wbuf0))
                                     (step-trace trace MT MA sigs
                                                 ISA spc smc)))
  :hints (("goal" :in-theory (disable member-equal)))))

(local
(defthm uniq-inst-at-LSU-wbuf0-MT-step-if-wbuf1-commit-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (uniq-inst-at-stg-in-trace '(commit wbuf1) trace)
                (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                (b1p (release-wbuf0? (MA-LSU MA) sigs))
                (b1p (wbuf-commit? (LSU-wbuf1 (MA-LSU MA)))))
           (uniq-inst-at-stgs-in-trace '((LSU wbuf0)
                                         (LSU wbuf0 lch)
                                         (complete wbuf0) (commit wbuf0))
                                       (step-trace trace MT MA sigs
                                                   ISA spc smc)))
  :hints (("goal" :in-theory (disable member-equal)))))

(defthm uniq-inst-at-LSU-wbuf0-MT-step-if-wbuf1-commit
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                (b1p (release-wbuf0? (MA-LSU MA) sigs))
                (b1p (wbuf-commit? (LSU-wbuf1 (MA-LSU MA)))))
           (uniq-inst-at-stgs '((LSU wbuf0) (LSU wbuf0 lch)
                                (complete wbuf0) (commit wbuf0))
```

```
                                (MT-step MT MA sigs)))
  :hints (("goal" :use (:instance UNIQ-INST-AT-STG-COMMIT-WBUF1)
                  :in-theory (enable uniq-inst-at-stg
                                     uniq-inst-at-stgs))))
)

(defthm member-equal-step-inst-wbuf0-if-not-release-wbuf0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
                  (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                  (not (b1p (flush-all? MA sigs))))
             (iff (member-equal (INST-stg (step-INST i MT MA sigs))
                                '((LSU wbuf0)
                                  (LSU wbuf0 lch)
                                  (complete wbuf0)
                                  (commit wbuf0)))
                  (member-equal (INST-stg i)
                                '((LSU wbuf0)
                                  (LSU wbuf0 lch)
                                  (complete wbuf0)
                                  (commit wbuf0)))))
  :hints (("goal" :in-theory (e/d (lift-b-ops release-wbuf0?
                                     check-wbuf1? flush-all?
                                     commit-stg-p
                                     step-inst-commit-inst
                                     step-inst-complete-inst
                                     complete-stg-p
                                     execute-stg-p LSU-STG-P
                                     step-inst-execute-inst
                                     INST-SELECT-WBUF0?
                                     RELEASE-WBUF0-READY?
                                     step-inst-low-level-functions)
                                  (inst-is-at-one-of-the-stages))
                  :use (:instance inst-is-at-one-of-the-stages))))

(encapsulate nil
(local
(defthm uniq-inst-at-LSU-wbuf0-MT-step-if-wbuf0-valid-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stgs-in-trace '((LSU wbuf0)
                                              (LSU wbuf0 lch)
                                              (complete wbuf0)
                                              (commit wbuf0))
                                            trace)
                  (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
                  (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                  (not (b1p (flush-all? MA sigs))))
             (no-inst-at-stgs-in-trace '((LSU wbuf0)
                                         (LSU wbuf0 lch)
                                         (complete wbuf0) (commit wbuf0))
                                       (step-trace trace MT MA sigs
                                                   ISA spc smc)))
  :hints (("goal" :in-theory (disable member-equal)))))

(local
(defthm uniq-inst-at-LSU-wbuf0-MT-step-if-wbuf0-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
```

```
                (MAETT-p MT) (MA-state-p MA)
                (uniq-inst-at-stgs-in-trace '((LSU wbuf0)
                                              (LSU wbuf0 lch)
                                              (complete wbuf0)
                                              (commit wbuf0))
                                            trace)
                (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
                (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                (not (b1p (flush-all? MA sigs))))
           (uniq-inst-at-stgs-in-trace '((LSU wbuf0)
                                         (LSU wbuf0 lch)
                                         (complete wbuf0) (commit wbuf0))
                                       (step-trace trace MT MA sigs
                                                   ISA spc smc)))
  :hints (("goal" :in-theory (disable member-equal)))))

(defthm uniq-inst-at-LSU-wbuf0-MT-step-if-wbuf0-valid
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
                (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                (not (b1p (flush-all? MA sigs))))
           (uniq-inst-at-stgs '((LSU wbuf0) (LSU wbuf0 lch)
                                (complete wbuf0) (commit wbuf0))
                              (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable uniq-inst-at-stgs)
                :use (:instance UNIQ-INST-AT-LSU-WBUF0-IF-VALID))))
)

(defthm not-member-equal-step-inst-wbuf0-if-release
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA)
                (not (equal (INST-stg i) '(commit wbuf0)))
                (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
                (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                (b1p (wbuf-commit? (LSU-wbuf0 (MA-LSU MA)))))
           (not (member-equal (INST-stg (step-INST i MT MA sigs))
                              '((LSU wbuf0)
                                (LSU wbuf0 lch)
                                (complete wbuf0) (commit wbuf0)))))
  :hints (("goal" :use (inst-is-at-one-of-the-stages)
                :in-theory (e/d (step-inst-low-level-functions
                                   commit-stg-p execute-stg-p
                                   complete-stg-p
                                   LSU-stg-p lift-b-ops
                                   INST-SELECT-WBUF0?
                                   step-inst-execute-inst
                                   step-inst-complete-inst
                                   step-inst-commit-inst)
                                 (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-LSU-wbuf0-MT-step-if-wbuf0-commit-help-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (no-inst-at-stg-in-trace '(commit wbuf0) trace)
                (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
                (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                (b1p (wbuf-commit? (LSU-wbuf0 (MA-LSU MA)))))
```

```
            (no-inst-at-stgs-in-trace '((LSU wbuf0)
                                       (LSU wbuf0 lch)
                                       (complete wbuf0) (commit wbuf0))
                                     (step-trace trace MT MA sigs
                                                 ISA spc smc)))
  :hints (("goal" :in-theory (disable member-equal)))))

(local
(defthm uniq-inst-at-LSU-wbuf0-MT-step-if-wbuf0-commit-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg-in-trace '(commit wbuf0) trace)
                  (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
                  (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                  (b1p (wbuf-commit? (LSU-wbuf0 (MA-LSU MA)))))
             (uniq-inst-at-stgs-in-trace '((LSU wbuf0)
                                           (LSU wbuf0 lch)
                                           (complete wbuf0) (commit wbuf0))
                                         (step-trace trace MT MA sigs
                                                     ISA spc smc)))
  :hints (("goal" :in-theory (disable member-equal)))))

(defthm uniq-inst-at-LSU-wbuf0-MT-step-if-wbuf0-commit
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
                  (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                  (b1p (wbuf-commit? (LSU-wbuf0 (MA-LSU MA)))))
             (uniq-inst-at-stgs '((LSU wbuf0) (LSU wbuf0 lch)
                                  (complete wbuf0) (commit wbuf0))
                                (MT-step MT MA sigs)))
  :hints (("goal" :use (:instance UNIQ-INST-AT-STG-COMMIT-WBUF0)
                  :in-theory (enable uniq-inst-at-stg uniq-inst-at-stgs))))
)

(defthm uniq-inst-at-LSU-wbuf0-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (wbuf-valid? (LSU-wbuf0 (step-LSU MA sigs)))))
             (uniq-inst-at-stgs '((LSU wbuf0) (LSU wbuf0 lch)
                                  (complete wbuf0) (commit wbuf0))
                                (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-LSU step-wbuf0
                                     WBUF1-OUTPUT UPDATE-WBUF0
                                     ISSUED-WRITE
                                     lift-b-ops))))

;; Proof of no-inst-at-LSU-wbuf0-MT-step
(encapsulate nil
(local
(defthm no-inst-at-stgs-nil-help
    (no-inst-at-stgs-in-trace nil trace)))

(defthm no-inst-at-stgs-nil
    (no-inst-at-stgs nil MT)
  :hints (("goal" :in-theory (enable no-inst-at-stgs))))
)

(encapsulate nil
(local
(defthm uniq-inst-at-stgs-nil-help
```

```
    (not (uniq-inst-at-stgs-in-trace nil trace))))

(defthm uniq-inst-at-stgs-nil
    (not (uniq-inst-at-stgs nil MT))
  :hints (("goal" :in-theory (enable uniq-inst-at-stgs))))
)

(encapsulate nil
(local
(defthm no-inst-at-stgs-singleton-help
    (iff (no-inst-at-stgs-in-trace (list stg1) trace)
         (no-inst-at-stg-in-trace stg1 trace))))

(defthm no-inst-at-stgs-singleton
    (iff (no-inst-at-stgs (list stg1) MT)
         (no-inst-at-stg stg1 MT))
  :hints (("goal" :in-theory (enable no-inst-at-stgs no-inst-at-stg))))
)

(encapsulate nil
(local
(defthm no-inst-at-stgs-if-no-inst-at-stg-help
    (implies (no-inst-at-stg-in-trace stg1 trace)
             (iff (no-inst-at-stgs-in-trace (cons stg1 stgs) trace)
                  (no-inst-at-stgs-in-trace stgs trace)))))

(defthm no-inst-at-stgs-if-no-inst-at-stg
    (implies (no-inst-at-stg stg1 MT)
             (iff (no-inst-at-stgs (cons stg1 stgs) MT)
                  (no-inst-at-stgs stgs MT)))
  :hints (("goal" :in-theory (enable no-inst-at-stgs no-inst-at-stg))))

(local
(defthm uniq-inst-at-stgs-if-no-inst-at-stg-help
    (implies (no-inst-at-stg-in-trace stg1 trace)
             (iff (uniq-inst-at-stgs-in-trace (cons stg1 stgs) trace)
                  (uniq-inst-at-stgs-in-trace stgs trace)))))

(defthm uniq-inst-at-stgs-if-no-inst-at-stg
    (implies (no-inst-at-stg stg1 MT)
             (iff (uniq-inst-at-stgs (cons stg1 stgs) MT)
                  (uniq-inst-at-stgs stgs MT)))
  :hints (("goal" :in-theory (enable uniq-inst-at-stgs
                                     no-inst-at-stg))))

)

(defthm  not-member-equal-step-inst-wbuf0-if-no-issue
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
                  (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                  (implies (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                           (not (b1p (LSU-RS-LD-st? (LSU-RS0 (MA-LSU MA))))))
                  (implies (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                           (not (b1p (LSU-RS-LD-st? (LSU-RS1 (MA-LSU MA)))))))
             (not (member-equal (INST-stg (step-INST i MT MA sigs))
                                '((LSU wbuf0)
                                  (LSU wbuf0 lch)
                                  (complete wbuf0) (commit wbuf0)))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
```

```
                  :in-theory (e/d (step-inst-low-level-functions
                                   step-inst-execute-inst
                                   step-inst-complete-inst
                                   step-inst-commit-inst
                                   complete-stg-p
                                   execute-stg-p LSU-stg-p
                                   commit-stg-p)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-LSU-wbuf0-MT-step-if-not-issue-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
                  (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                  (implies (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                           (not (b1p (LSU-RS-LD-st? (LSU-RS0 (MA-LSU MA))))))
                  (implies (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                           (not (b1p (LSU-RS-LD-st? (LSU-RS1 (MA-LSU MA)))))))
             (no-inst-at-stgs-in-trace '((LSU wbuf0)
                                         (LSU wbuf0 lch)
                                         (complete wbuf0) (commit wbuf0))
                                       (step-trace trace MT MA sigs
                                                   ISA spc smc)))
  :hints (("goal" :in-theory (disable member-equal)))))

(defthm no-inst-at-LSU-wbuf0-MT-step-if-not-issue
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
                  (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                  (implies (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                           (not (b1p (LSU-RS-LD-st? (LSU-RS0 (MA-LSU MA))))))
                  (implies (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                           (not (b1p (LSU-RS-LD-st? (LSU-RS1 (MA-LSU MA)))))))
             (no-inst-at-stgs '((LSU wbuf0) (LSU wbuf0 lch)
                                (complete wbuf0) (commit wbuf0))
                              (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stgs))))
)

(defthm  not-member-equal-step-inst-wbuf0-if-not-wbuf1-valid
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (release-wbuf0? (MA-LSU MA) sigs))
                  (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                  (implies (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                           (not (b1p (LSU-RS-LD-st? (LSU-RS0 (MA-LSU MA))))))
                  (implies (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                           (not (b1p (LSU-RS-LD-st? (LSU-RS1 (MA-LSU MA)))))))
             (not (member-equal (INST-stg (step-INST i MT MA sigs))
                                '((LSU wbuf0)
                                  (LSU wbuf0 lch)
                                  (complete wbuf0) (commit wbuf0)))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-inst-low-level-functions
                                   step-inst-execute-inst
                                   step-inst-complete-inst
                                   step-inst-commit-inst
```

```
                                   release-wbuf0?
                                   RELEASE-WBUF0-READY?
                                   complete-stg-p lift-b-ops
                                   execute-stg-p LSU-stg-p
                                   commit-stg-p)
                                (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-LSU-wbuf0-MT-step-if-not-wbuf1-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (release-wbuf0? (MA-LSU MA) sigs))
                  (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                  (implies (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                           (not (b1p (LSU-RS-LD-st? (LSU-RS0 (MA-LSU MA))))))
                  (implies (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                           (not (b1p (LSU-RS-LD-st? (LSU-RS1 (MA-LSU MA)))))))
             (no-inst-at-stgs-in-trace '((LSU wbuf0)
                                         (LSU wbuf0 lch)
                                         (complete wbuf0) (commit wbuf0))
                                       (step-trace trace MT MA sigs
                                                   ISA spc smc)))
  :hints (("goal" :in-theory (disable member-equal)))))

(defthm no-inst-at-LSU-wbuf0-MT-step-if-not-wbuf1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (release-wbuf0? (MA-LSU MA) sigs))
                  (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                  (implies (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                           (not (b1p (LSU-RS-LD-st? (LSU-RS0 (MA-LSU MA))))))
                  (implies (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                           (not (b1p (LSU-RS-LD-st? (LSU-RS1 (MA-LSU MA)))))))
             (no-inst-at-stgs '((LSU wbuf0) (LSU wbuf0 lch)
                                (complete wbuf0) (commit wbuf0))
                              (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stgs))))
)

(defthm not-INST-stg-step-inst-commit-wbuf0-if-not-wbuf-valid
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
                  (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))))
             (not (equal (INST-stg (step-INST i MT MA sigs))
                         '(commit wbuf0))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-inst-low-level-functions
                                   step-inst-complete-inst complete-stg-p
                                   commit-stg-p step-inst-commit-inst)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-commit-wbuf0-MT-step-if-not-wbuf-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))))
```

```
                (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))))
            (no-inst-at-stg-in-trace '(commit wbuf0)
                                     (step-trace trace MT MA sigs
                                                 ISA spc smc)))))

(defthm no-inst-at-commit-wbuf0-MT-step-if-not-wbuf-valid
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
                (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))))
            (no-inst-at-stg  '(commit wbuf0) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg))))
)

(defthm not-equal-commit-wbuf0-if-release-wbuf0
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                (b1p (release-wbuf0? (MA-LSU MA) sigs)))
            (not (equal (INST-stg (step-inst i MT MA sigs)) '(commit wbuf0))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                :in-theory (e/d (step-inst-low-level-functions
                                    step-inst-commit-inst
                                    step-inst-complete-inst
                                    complete-stg-p lift-b-ops
                                    release-wbuf0? RELEASE-WBUF0-READY?
                                    commit-stg-p)
                                (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-commit-wbuf0-MT-step-if-release-wbuf0-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                (b1p (release-wbuf0? (MA-LSU MA) sigs)))
            (no-inst-at-stg-in-trace  '(commit wbuf0)
                                     (step-trace trace MT MA sigs
                                                 ISA spc smc)))))

(defthm no-inst-at-commit-wbuf0-MT-step-if-release-wbuf0
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                (b1p (release-wbuf0? (MA-LSU MA) sigs)))
            (no-inst-at-stg  '(commit wbuf0) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg))))
)

(defthm not-INST-commit-if-commit-jmp
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (or (equal (INST-stg i) '(complete wbuf0))
                    (equal (INST-stg i) '(complete wbuf1)))
                (not (MT-CMI-p (MT-step MT MA sigs)))
                (b1p (commit-jmp? MA))
                (MAETT-p MT) (MA-state-p MA))
            (equal (INST-commit? i MA) 0))
  :hints (("goal" :in-theory (enable INST-commit? lift-b-ops
                                     commit-jmp? equal-b1p-converter)
```

```
                :cases ((b1p (INST-specultv? i))
                        (b1p (INST-modified? i))))))

(defthm not-INST-commit-if-leave-excpt
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (or (equal (INST-stg i) '(complete wbuf0))
                      (equal (INST-stg i) '(complete wbuf1)))
                  (b1p (leave-excpt? MA))
                  (MAETT-p MT) (MA-state-p MA))
             (equal (INST-commit? i MA) 0))
  :hints (("goal" :in-theory (enable INST-commit? lift-b-ops
                                     commit-jmp? equal-b1p-converter)
                  :cases ((b1p (INST-specultv? i))
                          (b1p (INST-modified? i))))))

(defthm not-INST-stg-step-INST-commit-wbuf0-if-flush-all
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
                  (not (b1p (wbuf-commit? (LSU-wbuf0 (MA-LSU MA)))))
                  (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                  (b1p (flush-all? MA sigs)))
             (not (equal (INST-stg (step-INST i MT MA sigs)) '(commit wbuf0))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :restrict ((NOT-ROB-EMPTY-IF-INST-IS-EXECUTED ((i i))))
                  :in-theory (e/d (step-inst-low-level-functions
                                     step-inst-complete-inst complete-stg-p
                                     lift-b-ops flush-all?
                                     step-inst-commit-inst commit-stg-p)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm no-inst-at-commit-wbuf0-MT-step-if-not-wbuf0-commit-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
                  (not (b1p (wbuf-commit? (LSU-wbuf0 (MA-LSU MA)))))
                  (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                  (b1p (flush-all? MA sigs)))
             (no-inst-at-stg-in-trace  '(commit wbuf0)
                                       (step-trace trace MT MA sigs
                                                   ISA spc smc)))))

(defthm no-inst-at-commit-wbuf0-MT-step-if-not-wbuf0-commit
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
                  (not (b1p (wbuf-commit? (LSU-wbuf0 (MA-LSU MA)))))
                  (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                  (b1p (flush-all? MA sigs)))
             (no-inst-at-stg  '(commit wbuf0) (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stg))))
)
```

```
(defthm no-equal-INST-stg-step-inst-commit-wbuf0-if-not-wbuf1-commit
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                  (not (b1p (wbuf-commit? (LSU-wbuf1 (MA-LSU MA)))))
                  (b1p (release-wbuf0? (MA-LSU MA) sigs))
                  (b1p (flush-all? MA sigs)))
             (not (equal (INST-stg (step-inst i MT MA sigs))
                         '(commit wbuf0))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :restrict ((NOT-ROB-EMPTY-IF-INST-IS-EXECUTED ((i i))))
                  :in-theory (e/d (step-inst-low-level-functions
                                     step-inst-complete-inst complete-stg-p
                                     lift-b-ops flush-all?
                                     step-inst-commit-inst commit-stg-p)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
 (defthm no-inst-at-commit-wbuf0-MT-step-if-not-wbuf1-commit-help
     (implies (and (inv MT MA)
                   (subtrace-p trace MT) (INST-listp trace)
                   (MAETT-p MT) (MA-state-p MA)
                   (not (MT-CMI-p (MT-step MT MA sigs)))
                   (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                   (not (b1p (wbuf-commit? (LSU-wbuf1 (MA-LSU MA)))))
                   (b1p (release-wbuf0? (MA-LSU MA) sigs))
                   (b1p (flush-all? MA sigs)))
              (no-inst-at-stg-in-trace  '(commit wbuf0)
                                        (step-trace trace MT MA sigs
                                                    ISA spc smc))))))

(defthm no-inst-at-commit-wbuf0-MT-step-if-not-wbuf1-commit
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (not (b1p (wbuf-commit? (LSU-wbuf1 (MA-LSU MA)))))
                  (b1p (release-wbuf0? (MA-LSU MA) sigs))
                  (b1p (flush-all? MA sigs)))
             (no-inst-at-stg  '(commit wbuf0) (MT-step MT MA sigs)))
  :Hints (("goal" :in-theory (enable no-inst-at-stg))))
)

(defthm no-inst-at-LSU-wbuf0-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (not (MT-CMI-p (MT-step MT MA sigs)))
                  (not (b1p (wbuf-valid? (LSU-wbuf0 (step-LSU MA sigs))))))
             (no-inst-at-stgs '((LSU wbuf0) (LSU wbuf0 lch)
                                (complete wbuf0) (commit wbuf0))
                              (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-LSU step-wbuf0
                                     WBUF1-OUTPUT UPDATE-WBUF0
                                     ISSUED-WRITE
                                     lift-b-ops))))

;; Proof of uniq-inst-at-LSU-wbuf1-MT-step
(encapsulate nil
(local
```

```
(defthm not-member-equal-step-INST-wbuf1-if-issue-RS0-help
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(LSU RS0)))
                  (execute-stg-p (INST-stg i))
                  (or (and (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                           (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
                           (not (b1p (release-wbuf0? (MA-LSU MA) sigs))))
                      (and (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                           (b1p (release-wbuf0? (MA-LSU MA) sigs))))
                  (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                  (b1p (LSU-RS-LD-ST? (LSU-RS0 (MA-LSU MA)))))
             (not (member-equal (INST-stg (step-INST i MT MA sigs))
                                '((LSU wbuf1) (LSU wbuf1 lch)
                                  (complete wbuf1) (commit wbuf1)))))
  :hints (("goal" :in-theory (e/d (step-inst-low-level-functions
                                     step-inst-execute-inst
                                     lift-b-ops RELEASE-WBUF0?
                                     RELEASE-WBUF0-READY? CHECK-WBUF1?
                                     ISSUE-LSU-RS0? ISSUE-LSU-RS1?
                                     LSU-RS0-ISSUE-READY?
                                     LSU-RS1-ISSUE-READY?
                                     INST-SELECT-WBUF0?
                                     LSU-stg-p execute-stg-p)
                                   (inst-is-at-one-of-the-stages))
                  :use (:instance inst-is-at-one-of-the-stages)))))

(defthm not-member-equal-step-INST-wbuf1-if-issue-RS0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(LSU RS0)))
                  (or (and (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                           (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
                           (not (b1p (release-wbuf0? (MA-LSU MA) sigs))))
                      (and (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                           (b1p (release-wbuf0? (MA-LSU MA) sigs))))
                  (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                  (b1p (LSU-RS-LD-ST? (LSU-RS0 (MA-LSU MA)))))
             (not (member-equal (INST-stg (step-INST i MT MA sigs))
                                '((LSU wbuf1) (LSU wbuf1 lch)
                                  (complete wbuf1) (commit wbuf1)))))
  :hints (("goal" :in-theory (e/d (step-inst-low-level-functions
                                     step-inst-commit-inst
                                     step-inst-complete-inst
                                     lift-b-ops RELEASE-WBUF0?
                                     complete-stg-p
                                     commit-stg-p)
                                   (inst-is-at-one-of-the-stages))
                  :use (:instance inst-is-at-one-of-the-stages))))
)

(encapsulate nil
(local
(defthm uniq-inst-at-wbuf1-MT-step-if-issue-RS0-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(LSU RS0) trace)
                  (or (and (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                           (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
```

```
                          (not (b1p (release-wbuf0? (MA-LSU MA) sigs))))
                     (and (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                          (b1p (release-wbuf0? (MA-LSU MA) sigs))))
                (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                (b1p (LSU-RS-LD-ST? (LSU-RS0 (MA-LSU MA)))))
           (no-inst-at-stgs-in-trace '((LSU wbuf1)
                                       (LSU wbuf1 lch)
                                       (complete wbuf1) (commit wbuf1))
                                     (step-trace trace MT MA sigs
                                                 ISA spc smc)))
  :hints (("goal" :in-theory (e/d (INST-SELECT-WBUF0? lift-b-ops)
                                  (member-equal))))))

(local
(defthm uniq-inst-at-wbuf1-MT-step-if-issue-RS0-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg-in-trace '(LSU RS0) trace)
                  (or (and (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                           (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
                           (not (b1p (release-wbuf0? (MA-LSU MA) sigs))))
                      (and (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                           (b1p (release-wbuf0? (MA-LSU MA) sigs))))
                  (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                  (b1p (LSU-RS-LD-ST? (LSU-RS0 (MA-LSU MA))))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stgs-in-trace '((LSU wbuf1)
                                           (LSU wbuf1 lch)
                                           (complete wbuf1) (commit wbuf1))
                                         (step-trace trace MT MA sigs
                                                     ISA spc smc)))
  :hints (("goal" :in-theory (e/d (INST-SELECT-WBUF0? lift-b-ops)
                                  (member-equal))))))

(defthm uniq-inst-at-wbuf1-MT-step-if-issue-RS0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (or (and (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                           (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
                           (not (b1p (release-wbuf0? (MA-LSU MA) sigs))))
                      (and (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                           (b1p (release-wbuf0? (MA-LSU MA) sigs))))
                  (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                  (b1p (LSU-RS-LD-ST? (LSU-RS0 (MA-LSU MA))))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stgs '((LSU wbuf1) (LSU wbuf1 lch)
                                  (complete wbuf1) (commit wbuf1))
                                (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (e/d (uniq-inst-at-stgs uniq-inst-at-stg
                                     issue-LSU-RS0? LSU-RS0-ISSUE-READY?
                                     lift-b-ops)
                                  (member-equal))
                  :use (:instance UNIQ-INST-AT-LSU-RS0-IF-VALID))))
)

(encapsulate nil
(local
(defthm not-member-equal-step-INST-wbuf1-if-issue-RS1-help
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
```

```
                  (not (equal (INST-stg i) '(LSU RS1)))
                  (execute-stg-p (INST-stg i))
                  (or (and (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                           (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
                           (not (b1p (release-wbuf0? (MA-LSU MA) sigs))))
                      (and (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                           (b1p (release-wbuf0? (MA-LSU MA) sigs))))
                  (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                  (b1p (LSU-RS-LD-ST? (LSU-RS1 (MA-LSU MA)))))
             (not (member-equal (INST-stg (step-INST i MT MA sigs))
                                '((LSU wbuf1) (LSU wbuf1 lch)
                                  (complete wbuf1) (commit wbuf1)))))
  :hints (("goal" :in-theory (e/d (step-inst-low-level-functions
                                   step-inst-execute-inst
                                   lift-b-ops RELEASE-WBUF0?
                                   RELEASE-WBUF0-READY? CHECK-WBUF1?
                                   ISSUE-LSU-RS0? ISSUE-LSU-RS1?
                                   LSU-RS0-ISSUE-READY?
                                   LSU-RS1-ISSUE-READY?
                                   INST-SELECT-WBUF0?
                                   LSU-stg-p execute-stg-p)
                                  (inst-is-at-one-of-the-stages))
                  :use (:instance inst-is-at-one-of-the-stages)))))

(defthm not-member-equal-step-INST-wbuf1-if-issue-RS1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(LSU RS1)))
                  (or (and (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                           (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
                           (not (b1p (release-wbuf0? (MA-LSU MA) sigs))))
                      (and (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                           (b1p (release-wbuf0? (MA-LSU MA) sigs))))
                  (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                  (b1p (LSU-RS-LD-ST? (LSU-RS1 (MA-LSU MA)))))
             (not (member-equal (INST-stg (step-INST i MT MA sigs))
                                '((LSU wbuf1) (LSU wbuf1 lch)
                                  (complete wbuf1) (commit wbuf1)))))
  :hints (("goal" :in-theory (e/d (step-inst-low-level-functions
                                   step-inst-commit-inst
                                   step-inst-complete-inst
                                   lift-b-ops RELEASE-WBUF0?
                                   complete-stg-p
                                   commit-stg-p)
                                  (inst-is-at-one-of-the-stages))
                  :use (:instance inst-is-at-one-of-the-stages))))
)

(encapsulate nil
(local
(defthm uniq-inst-at-wbuf1-MT-step-if-issue-RS1-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(LSU RS1) trace)
                  (or (and (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                           (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
                           (not (b1p (release-wbuf0? (MA-LSU MA) sigs))))
                      (and (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                           (b1p (release-wbuf0? (MA-LSU MA) sigs))))
                  (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
```

```
                (b1p (LSU-RS-LD-ST? (LSU-RS1 (MA-LSU MA)))))
            (no-inst-at-stgs-in-trace '((LSU wbuf1)
                                       (LSU wbuf1 lch)
                                       (complete wbuf1) (commit wbuf1))
                                     (step-trace trace MT MA sigs
                                                 ISA spc smc)))
  :hints (("goal" :in-theory (e/d (INST-SELECT-WBUF0? lift-b-ops)
                                  (member-equal))))))

(local
(defthm uniq-inst-at-wbuf1-MT-step-if-issue-RS1-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg-in-trace '(LSU RS1) trace)
                  (or (and (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                           (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
                           (not (b1p (release-wbuf0? (MA-LSU MA) sigs))))
                      (and (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                           (b1p (release-wbuf0? (MA-LSU MA) sigs))))
                  (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                  (b1p (LSU-RS-LD-ST? (LSU-RS1 (MA-LSU MA))))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stgs-in-trace '((LSU wbuf1)
                                           (LSU wbuf1 lch)
                                           (complete wbuf1) (commit wbuf1))
                                         (step-trace trace MT MA sigs
                                                     ISA spc smc)))
  :hints (("goal" :in-theory (e/d (INST-SELECT-WBUF0? lift-b-ops)
                                  (member-equal))))))

(defthm uniq-inst-at-wbuf1-MT-step-if-issue-RS1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (or (and (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                           (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA))))
                           (not (b1p (release-wbuf0? (MA-LSU MA) sigs))))
                      (and (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                           (b1p (release-wbuf0? (MA-LSU MA) sigs))))
                  (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                  (b1p (LSU-RS-LD-ST? (LSU-RS1 (MA-LSU MA))))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stgs '((LSU wbuf1) (LSU wbuf1 lch)
                                  (complete wbuf1) (commit wbuf1))
                                (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (e/d (uniq-inst-at-stgs uniq-inst-at-stg
                                     issue-LSU-RS1? LSU-RS1-ISSUE-READY?
                                     lift-b-ops)
                                  (member-equal))
                  :use (:instance UNIQ-INST-AT-LSU-RS1-IF-VALID))))
)

(defthm not-member-equal-step-INST-wbuf1-if-not-at-wbuf1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                  (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                  (not (b1p (flush-all? MA sigs))))
             (iff (member-equal (INST-stg (step-INST i MT MA sigs))
                                '((LSU wbuf1)
                                  (LSU wbuf1 lch)
```

```
                                (complete wbuf1) (commit wbuf1)))
                (member-equal (INST-stg i)
                              '((LSU wbuf1)
                                (LSU wbuf1 lch)
                                (complete wbuf1)
                                (commit wbuf1)))))
  :hints (("goal" :in-theory (e/d (step-inst-low-level-functions
                                   step-inst-commit-inst commit-stg-p
                                   complete-stg-p step-inst-complete-inst
                                   lift-b-ops flush-all?
                                   INST-SELECT-WBUF0? LSU-stg-p
                                   ISSUE-LSU-RS0? ISSUE-LSU-RS1?
                                   step-inst-execute-inst execute-stg-p)
                                  (inst-is-at-one-of-the-stages))
                :use (:instance inst-is-at-one-of-the-stages))))


(encapsulate nil
(local
(defthm uniq-inst-at-wbuf1-MT-step-if-wbuf1-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stgs-in-trace '((LSU wbuf1)
                                              (LSU wbuf1 lch)
                                              (complete wbuf1)
                                              (commit wbuf1))
                                            trace)
                  (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                  (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                  (not (b1p (flush-all? MA sigs))))
             (no-inst-at-stgs-in-trace '((LSU wbuf1)
                                         (LSU wbuf1 lch)
                                         (complete wbuf1) (commit wbuf1))
                                       (step-trace trace MT MA sigs
                                                   ISA spc smc)))
  :hints (("goal" :in-theory (disable member-equal)))))

(local
(defthm uniq-inst-at-wbuf1-MT-step-if-wbuf1-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stgs-in-trace '((LSU wbuf1)
                                                (LSU wbuf1 lch)
                                                (complete wbuf1)
                                                (commit wbuf1))
                                              trace)
                  (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                  (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                  (not (b1p (flush-all? MA sigs))))
             (uniq-inst-at-stgs-in-trace '((LSU wbuf1)
                                           (LSU wbuf1 lch)
                                           (complete wbuf1) (commit wbuf1))
                                         (step-trace trace MT MA sigs
                                                     ISA spc smc)))
  :hints (("goal" :in-theory (disable member-equal)))))

(defthm uniq-inst-at-wbuf1-MT-step-if-wbuf1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
```

```
                (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                (not (b1p (flush-all? MA sigs))))
           (uniq-inst-at-stgs '((LSU wbuf1) (LSU wbuf1 lch)
                                (complete wbuf1) (commit wbuf1))
                              (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable uniq-inst-at-stgs)
                  :use (:instance UNIQ-INST-AT-LSU-WBUF1-IF-VALID))))
)

(defthm not-INST-stg-step-inst-wbuf1-if-wbuf1-valid
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)N
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(commit wbuf1)))
                  (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                  (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                  (b1p (wbuf-commit? (LSU-wbuf1 (MA-LSU MA)))))
             (not (member-equal (INST-stg (step-inst i MT MA sigs))
                                '((LSU wbuf1)
                                  (LSU wbuf1 lch)
                                  (complete wbuf1) (commit wbuf1)))))
  :hints (("goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (e/d (step-inst-low-level-functions
                                     step-inst-commit-inst
                                     step-inst-execute-inst
                                     step-inst-complete-inst
                                     complete-stg-p execute-stg-p
                                     LSU-stg-p issue-LSU-RS1?
                                     issue-LSU-RS0? lift-b-ops
                                     commit-stg-p)
                                  (inst-is-at-one-of-the-stages)))))

(encapsulate nil
(local
(defthm uniq-inst-at-wbuf1-MT-step-if-wbuf1-commit-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(commit wbuf1) trace)
                  (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                  (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                  (b1p (wbuf-commit? (LSU-wbuf1 (MA-LSU MA)))))
             (no-inst-at-stgs-in-trace '((LSU wbuf1)
                                         (LSU wbuf1 lch)
                                         (complete wbuf1) (commit wbuf1))
                                       (step-trace trace MT MA sigs
                                                   ISA spc smc)))
  :hints (("goal" :in-theory (disable member-equal)))))

(local
(defthm uniq-inst-at-wbuf1-MT-step-if-wbuf1-commit-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg-in-trace '(commit wbuf1) trace)
                  (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                  (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                  (b1p (wbuf-commit? (LSU-wbuf1 (MA-LSU MA)))))
             (uniq-inst-at-stgs-in-trace '((LSU wbuf1)
                                           (LSU wbuf1 lch)
                                           (complete wbuf1) (commit wbuf1))
                                         (step-trace trace MT MA sigs
```

```
                                                        ISA spc smc)))
  :hints (("goal" :in-theory (disable member-equal)))))

(defthm uniq-inst-at-wbuf1-MT-step-if-wbuf1-commit
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                  (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                  (b1p (wbuf-commit? (LSU-wbuf1 (MA-LSU MA)))))
             (uniq-inst-at-stgs '((LSU wbuf1) (LSU wbuf1 lch)
                                  (complete wbuf1) (commit wbuf1))
                                (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable uniq-inst-at-stgs
                                     uniq-inst-at-stg)
                  :use (:instance UNIQ-INST-AT-STG-COMMIT-WBUF1))))
)

(defthm uniq-inst-at-LSU-wbuf1-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (wbuf-valid? (LSU-wbuf1 (step-LSU MA sigs)))))
             (uniq-inst-at-stgs '((LSU wbuf1) (LSU wbuf1 lch)
                                  (complete wbuf1) (commit wbuf1))
                                (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-LSU step-wbuf1 lift-b-ops
                                     update-wbuf1 ISSUED-WRITE))))

;; Proof of no-inst-at-LSU-wbuf1-MT-step
(defthm not-member-equal-step-inst-wbuf1-if-wbuf-empty
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
                  (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))))
             (not (member-equal (INST-stg (step-INST i MT MA sigs))
                                '((LSU wbuf1) (LSU wbuf1 lch)
                                  (complete wbuf1) (commit wbuf1)))))
  :hints (("goal" :in-theory (e/d (step-inst-low-level-functions
                                     commit-stg-p step-inst-commit-inst
                                     step-inst-execute-inst execute-stg-p
                                     LSU-stg-p lift-b-ops
                                     INST-SELECT-WBUF0?
                                     complete-stg-p step-inst-complete-inst)
                                  (inst-is-at-one-of-the-stages))
                  :use (:instance inst-is-at-one-of-the-stages))))

(encapsulate nil
(local
(defthm no-inst-at-LSU-wbuf1-MT-step-if-not-wbuf-empty-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
                  (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))))
             (no-inst-at-stgs-in-trace '((LSU wbuf1)
                                         (LSU wbuf1 lch)
                                         (complete wbuf1) (commit wbuf1))
                                       (step-trace trace MT MA sigs
                                                   ISA spc smc)))
  :hints (("goal" :in-theory (disable member-equal)))))

(defthm no-inst-at-LSU-wbuf1-MT-step-if-not-wbuf-empty
```

```
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (wbuf-valid? (LSU-wbuf0 (MA-LSU MA)))))
                  (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))))
             (no-inst-at-stgs '((LSU wbuf1) (LSU wbuf1 lch)
                                (complete wbuf1) (commit wbuf1))
                              (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable no-inst-at-stgs))))
)

(defthm not-member-equal-step-inst-wbuf1-if-not-issue
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (member-equal (INST-stg i)
                                     '((LSU wbuf1)
                                       (LSU wbuf1 lch)
                                       (complete wbuf1) (commit wbuf1))))
                  (implies (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                           (not (b1p (LSU-RS-LD-st? (LSU-RS0 (MA-LSU MA))))))
                  (implies (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                           (not (b1p (LSU-RS-LD-st? (LSU-RS1 (MA-LSU MA)))))))
             (not (member-equal (INST-stg (step-INST i MT MA sigs))
                                '((LSU wbuf1)
                                  (LSU wbuf1 lch)
                                  (complete wbuf1) (commit wbuf1)))))
  :hints (("Goal" :in-theory (e/d (step-inst-low-level-functions
                                   step-inst-commit-inst
                                   step-inst-execute-inst
                                   step-inst-complete-inst
                                   complete-stg-p
                                   execute-stg-p LSU-stg-p
                                   commit-stg-p)
                                  (inst-is-at-one-of-the-stages))
                  :use (inst-is-at-one-of-the-stages))))

(encapsulate nil
(local
(defthm no-inst-at-LSU-wbuf1-MT-step-if-no-issue-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stgs-in-trace '((LSU wbuf1)
                                              (LSU wbuf1 lch)
                                              (complete wbuf1)
                                              (commit wbuf1))
                                            trace)
                  (implies (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                           (not (b1p (LSU-RS-LD-st? (LSU-RS0 (MA-LSU MA))))))
                  (implies (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                           (not (b1p (LSU-RS-LD-st? (LSU-RS1 (MA-LSU MA)))))))
             (no-inst-at-stgs-in-trace '((LSU wbuf1)
                                         (LSU wbuf1 lch)
                                         (complete wbuf1) (commit wbuf1))
                                       (step-trace trace MT MA sigs
                                                   ISA spc smc)))
  :hints (("Goal" :in-theory (disable member-equal)))))

(defthm no-inst-at-LSU-wbuf1-MT-step-if-no-issue
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
```

```
                (implies (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                         (not (b1p (LSU-RS-LD-st? (LSU-RS0 (MA-LSU MA))))))
                (implies (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                         (not (b1p (LSU-RS-LD-st? (LSU-RS1 (MA-LSU MA)))))))
           (no-inst-at-stgs '((LSU wbuf1) (LSU wbuf1 lch)
                              (complete wbuf1) (commit wbuf1))
                            (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable no-inst-at-stgs)
                  :use (:instance no-inst-at-LSU-wbuf1))))
)

(defthm not-member-equal-step-inst-wbuf1-if-release-wbuf0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                  (b1p (release-wbuf0? (MA-LSU MA) sigs)))
           (not (member-equal (INST-stg (step-INST i MT MA sigs))
                              '((LSU wbuf1)
                                (LSU wbuf1 lch)
                                (complete wbuf1)
                                (commit wbuf1))))))
  :hints (("Goal" :in-theory (e/d (step-inst-low-level-functions
                                   step-inst-commit-inst
                                   step-inst-complete-inst
                                   step-inst-execute-inst
                                   execute-stg-p LSU-stg-p
                                   INST-SELECT-WBUF0? lift-b-ops
                                   commit-stg-p complete-stg-p)
                                  (inst-is-at-one-of-the-stages))
                  :use (inst-is-at-one-of-the-stages))))

(encapsulate nil
(local
(defthm no-inst-at-LSU-wbuf1-MT-step-if-not-wbuf1-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                  (b1p (release-wbuf0? (MA-LSU MA) sigs)))
           (no-inst-at-stgs-in-trace '((LSU wbuf1)
                                       (LSU wbuf1 lch)
                                       (complete wbuf1)
                                       (commit wbuf1))
                                     (step-trace trace MT MA sigs ISA
                                                 spc smc)))
  :hints (("Goal" :in-theory (disable member-equal)))))

(defthm no-inst-at-LSU-wbuf1-MT-step-if-not-wbuf1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA)))))
                  (b1p (release-wbuf0? (MA-LSU MA) sigs)))
           (no-inst-at-stgs '((LSU wbuf1) (LSU wbuf1 lch)
                              (complete wbuf1) (commit wbuf1))
                            (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable no-inst-at-stgs))))
)

(defthm not-member-equal-step-inst-wbuf1-if-not-issue-release-wbuf0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
```

```
                (MAETT-p MT) (MA-state-p MA)
                (implies (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                         (not (b1p (LSU-RS-LD-st? (LSU-RS0 (MA-LSU MA))))))
                (implies (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                         (not (b1p (LSU-RS-LD-st? (LSU-RS1 (MA-LSU MA))))))
                (b1p (release-wbuf0? (MA-LSU MA) sigs)))
           (not (member-equal  (INST-stg (step-INST i MT MA sigs))
                               '((LSU wbuf1)
                                 (LSU wbuf1 lch)
                                 (complete wbuf1) (commit wbuf1)))))
    :hints (("Goal" :in-theory (e/d (step-inst-low-level-functions
                                     step-inst-commit-inst
                                     step-inst-complete-inst
                                     step-inst-execute-inst
                                     execute-stg-p LSU-stg-p
                                     RELEASE-WBUF0?
                                     INST-SELECT-WBUF0? lift-b-ops
                                     commit-stg-p complete-stg-p)
                                    (inst-is-at-one-of-the-stages))
                    :use (inst-is-at-one-of-the-stages))))

(encapsulate nil
(local
(defthm no-inst-at-LSU-wbuf1-MT-step-if-release-wbuf0-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (implies (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                           (not (b1p (LSU-RS-LD-st? (LSU-RS0 (MA-LSU MA))))))
                  (implies (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                           (not (b1p (LSU-RS-LD-st? (LSU-RS1 (MA-LSU MA))))))
                  (b1p (release-wbuf0? (MA-LSU MA) sigs)))
             (no-inst-at-stgs-in-trace '((LSU wbuf1)
                                         (LSU wbuf1 lch)
                                         (complete wbuf1) (commit wbuf1))
                                       (step-trace trace MT MA sigs
                                                   ISA spc smc)))
  :hints (("Goal" :in-theory (disable member-equal)))))

(defthm no-inst-at-LSU-wbuf1-MT-step-if-release-wbuf0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (implies (b1p (issue-LSU-RS0? (MA-LSU MA) MA sigs))
                           (not (b1p (LSU-RS-LD-st? (LSU-RS0 (MA-LSU MA))))))
                  (implies (b1p (issue-LSU-RS1? (MA-LSU MA) MA sigs))
                           (not (b1p (LSU-RS-LD-st? (LSU-RS1 (MA-LSU MA))))))
                  (b1p (release-wbuf0? (MA-LSU MA) sigs)))
             (no-inst-at-stgs '((LSU wbuf1) (LSU wbuf1 lch)
                                (complete wbuf1) (commit wbuf1))
                              (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable no-inst-at-stgs))))
)

(defthm not-INST-stg-step-inst-commit-wbuf-if-release-wbuf0
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (release-wbuf0? (MA-LSU MA) sigs)))
             (not (equal (INST-stg (step-INST i MT MA sigs)) '(commit wbuf1))))
  :hints (("Goal" :in-theory (e/d (step-inst-low-level-functions
                                   step-inst-commit-inst commit-stg-p
                                   lift-b-ops
```

```
                                    step-inst-complete-inst complete-stg-p)
                                   (inst-is-at-one-of-the-stages))
                  :use (inst-is-at-one-of-the-stages))))

(encapsulate nil
(local
(defthm no-inst-at-commit-wbuf1-MT-step-if-release-wbuf0-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (release-wbuf0? (MA-LSU MA) sigs)))
             (no-inst-at-stg-in-trace '(commit wbuf1)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(defthm no-inst-at-commit-wbuf1-MT-step-if-release-wbuf0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (release-wbuf0? (MA-LSU MA) sigs)))
             (no-inst-at-stg '(commit wbuf1) (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable no-inst-at-stg))))
)

(defthm not-equal-INST-stg-step-inst-commit-wbuf1-if-not-wbuf1-valid
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))))
             (not (equal (INST-stg (step-INST i MT MA sigs)) '(commit wbuf1))))
  :hints (("Goal" :in-theory (e/d (step-inst-low-level-functions
                                     step-inst-commit-inst
                                     step-inst-complete-inst
                                     commit-stg-p complete-stg-p)
                                   (inst-is-at-one-of-the-stages))
                  :use (:instance inst-is-at-one-of-the-stages))))

(encapsulate nil
(local
(defthm no-inst-at-commit-wbuf1-MT-step-if-not-wbuf1-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))))
             (no-inst-at-stg-in-trace '(commit wbuf1)
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))))

(defthm no-inst-at-commit-wbuf1-MT-step-if-not-wbuf1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))))
             (no-inst-at-stg '(commit wbuf1) (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable no-inst-at-stg))))
)

(defthm not-INST-stg-step-inst-commit-wbuf1-if-flush-all
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                  (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                  (not (b1p (wbuf-commit? (LSU-wbuf1 (MA-LSU MA)))))
```

```
                (not (MT-CMI-p (MT-step MT MA sigs)))
                (b1p (flush-all? MA sigs)))
           (not (equal (INST-stg (step-INST i MT MA sigs)) '(commit wbuf1))))
  :hints (("Goal" :in-theory (e/d (step-inst-low-level-functions
                                   step-inst-commit-inst commit-stg-p
                                   step-inst-complete-inst complete-stg-p
                                   lift-b-ops FLUSH-ALL?)
                                  (inst-is-at-one-of-the-stages))
                :restrict ((NOT-ROB-EMPTY-IF-INST-IS-EXECUTED ((i i))))
                :use (inst-is-at-one-of-the-stages))))

(encapsulate nil
(local
(defthm no-inst-at-commit-wbuf1-MT-step-if-not-wbuf1-commit-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                (not (b1p (wbuf-commit? (LSU-wbuf1 (MA-LSU MA)))))
                (not (MT-CMI-p (MT-step MT MA sigs)))
                (b1p (flush-all? MA sigs)))
           (no-inst-at-stg-in-trace '(commit wbuf1)
                                    (step-trace trace MT MA sigs
                                                ISA spc smc)))))

(defthm no-inst-at-commit-wbuf1-MT-step-if-not-wbuf1-commit
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (b1p (wbuf-valid? (LSU-wbuf1 (MA-LSU MA))))
                (not (b1p (release-wbuf0? (MA-LSU MA) sigs)))
                (not (b1p (wbuf-commit? (LSU-wbuf1 (MA-LSU MA)))))
                (not (MT-CMI-p (MT-step MT MA sigs)))
                (b1p (flush-all? MA sigs)))
           (no-inst-at-stg '(commit wbuf1) (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable no-inst-at-stg))))
)

(defthm no-inst-at-LSU-wbuf1-MT-step
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                (not (MT-CMI-p (MT-step MT MA sigs)))
                (not (b1p (wbuf-valid? (LSU-wbuf1 (step-LSU MA sigs))))))
           (no-inst-at-stgs '((LSU wbuf1) (LSU wbuf1 lch)
                              (complete wbuf1) (commit wbuf1))
                            (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (enable step-LSU step-wbuf1 lift-b-ops
                                     issued-write UPDATE-WBUF1))))

;; Proof of uniq-inst-at-LSU-lch-MT-step
(defthm not-member-equal-step-INST-LSU-lch-if-check-wbuf0
    (implies (and (inv MT MA)
                (INST-in i MT) (INST-p i)
                (MAETT-p MT) (MA-state-p MA)
                (not (equal (INST-stg i) '(LSU wbuf0)))
                (b1p (check-wbuf0? (MA-LSU MA))))
           (not (member-equal (INST-stg (step-INST i MT MA sigs))
                              '((LSU lch) (LSU wbuf0 lch)
                                (LSU wbuf1 lch)))))
  :hints (("Goal" :in-theory (e/d (step-inst-low-level-functions
                                   step-inst-execute-inst
                                   CHECK-WBUF1? lift-b-ops
```

```
                                   RELEASE-RBUF?
                                   execute-stg-p LSU-stg-p)
                                (inst-is-at-one-of-the-stages))
                  :use (:instance inst-is-at-one-of-the-stages))))

(encapsulate nil
(local
(defthm uniq-inst-at-LSU-lch-MT-step-if-check-wbuf0-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(LSU wbuf0) trace)
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (check-wbuf0? (MA-LSU MA))))
             (no-inst-at-stgs-in-trace '((LSU lch)
                                         (LSU wbuf0 lch)
                                         (LSU wbuf1 lch))
                                       (step-trace trace MT MA sigs
                                                   ISA spc smc)))
  :hints (("Goal" :in-theory (disable member-equal)))))

(local
(defthm uniq-inst-at-LSU-lch-MT-step-if-check-wbuf0-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg-in-trace '(LSU wbuf0) trace)
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (check-wbuf0? (MA-LSU MA))))
             (uniq-inst-at-stgs-in-trace '((LSU lch)
                                           (LSU wbuf0 lch)
                                           (LSU wbuf1 lch))
                                         (step-trace trace MT MA sigs
                                                     ISA spc smc)))
  :hints (("Goal" :in-theory (disable member-equal)))))

(defthm uniq-inst-at-LSU-lch-MT-step-if-check-wbuf0
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (check-wbuf0? (MA-LSU MA))))
             (uniq-inst-at-stgs '((LSU lch) (LSU wbuf0 lch)
                                  (LSU wbuf1 lch))
                                (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable uniq-inst-at-stgs uniq-inst-at-stg
                                     check-wbuf0? lift-b-ops)
                  :use (:instance uniq-inst-at-stg-LSU-wbuf0))))
)

(defthm not-member-equal-step-INST-if-check-wbuf1
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(LSU wbuf1)))
                  (b1p (check-wbuf1? (MA-LSU MA))))
             (not (member-equal (INST-stg (step-INST i MT MA sigs))
                                '((LSU lch)
                                  (LSU wbuf0 lch)
                                  (LSU wbuf1 lch)))))
  :hints (("Goal" :in-theory (e/d (step-inst-low-level-functions
                                   step-inst-execute-inst
                                   CHECK-WBUF1? lift-b-ops
```

```
                                   RELEASE-RBUF?
                                   execute-stg-p LSU-stg-p)
                                 (inst-is-at-one-of-the-stages))
                 :use (:instance inst-is-at-one-of-the-stages))))

(encapsulate nil
(local
(defthm uniq-inst-at-LSU-lch-MT-step-if-check-wbuf1-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(LSU wbuf1) trace)
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (check-wbuf1? (MA-LSU MA))))
             (no-inst-at-stgs-in-trace '((LSU lch)
                                         (LSU wbuf0 lch)
                                         (LSU wbuf1 lch))
                                       (step-trace trace MT MA sigs
                                                   ISA spc smc)))
  :hints (("Goal" :in-theory (disable member-equal)))))

(local
(defthm uniq-inst-at-LSU-lch-MT-step-if-check-wbuf1-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg-in-trace '(LSU wbuf1) trace)
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (check-wbuf1? (MA-LSU MA))))
             (uniq-inst-at-stgs-in-trace '((LSU lch)
                                           (LSU wbuf0 lch)
                                           (LSU wbuf1 lch))
                                         (step-trace trace MT MA sigs
                                                     ISA spc smc)))
  :hints (("Goal" :in-theory (disable member-equal)))))

(defthm uniq-inst-at-LSU-lch-MT-step-if-check-wbuf1
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (check-wbuf1? (MA-LSU MA))))
             (uniq-inst-at-stgs '((LSU lch) (LSU wbuf0 lch)
                                  (LSU wbuf1 lch))
                                (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable uniq-inst-at-stgs uniq-inst-at-stg
                                     check-wbuf1? lift-b-ops)
                  :use (:instance uniq-inst-at-stg-LSU-wbuf1))))
)

(defthm not-member-equal-step-inst-LSU-lch-if-release-rbuf
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(LSU rbuf)))
                  (b1p (release-rbuf? (MA-LSU MA) MA sigs)))
             (not (member-equal (INST-stg (step-INST i MT MA sigs))
                                '((LSU lch)
                                  (LSU wbuf0 lch)
                                  (LSU wbuf1 lch)))))
  :hints (("Goal" :in-theory (e/d (step-inst-execute-inst
                                   step-inst-low-level-functions
                                   execute-stg-p LSU-stg-p
```

```
                                   RELEASE-RBUF? lift-b-ops)
                                 (inst-is-at-one-of-the-stages))
                  :use (:instance inst-is-at-one-of-the-stages))))

(encapsulate nil
(local
(defthm uniq-inst-at-LSU-lch-MT-step-if-release-rbuf-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (no-inst-at-stg-in-trace '(LSU rbuf) trace)
                  (b1p (release-rbuf? (MA-LSU MA) MA sigs)))
             (no-inst-at-stgs-in-trace '((LSU lch)
                                         (LSU wbuf0 lch)
                                         (LSU wbuf1 lch))
                                       (step-trace trace MT MA sigs
                                                   ISA spc smc)))
  :hints (("Goal" :in-theory (disable member-equal)))))

(local
(defthm uniq-inst-at-LSU-lch-MT-step-if-release-rbuf-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (uniq-inst-at-stg-in-trace '(LSU rbuf) trace)
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (release-rbuf? (MA-LSU MA) MA sigs)))
             (uniq-inst-at-stgs-in-trace '((LSU lch)
                                           (LSU wbuf0 lch)
                                           (LSU wbuf1 lch))
                                         (step-trace trace MT MA sigs
                                                     ISA spc smc)))
  :hints (("Goal" :in-theory (disable member-equal)))))

(defthm uniq-inst-at-LSU-lch-MT-step-if-release-rbuf
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (release-rbuf? (MA-LSU MA) MA sigs)))
             (uniq-inst-at-stgs '((LSU lch) (LSU wbuf0 lch)
                                  (LSU wbuf1 lch))
                                (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable uniq-inst-at-stgs uniq-inst-at-stg
                                     release-rbuf? lift-b-ops)
                  :use (:instance uniq-inst-at-LSU-rbuf-if-valid))))
)

(defthm uniq-inst-at-LSU-lch-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (LSU-latch-valid? (LSU-lch (step-LSU MA sigs)))))
             (uniq-inst-at-stgs '((LSU lch) (LSU wbuf0 lch)
                                  (LSU wbuf1 lch))
                                (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable step-LSU step-LSU-lch lift-b-ops))))

;; Proof of no-inst-at-LSU-lch-MT-step
(defthm not-member-equal-step-INST-LSU-lch-if-no-check-release
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (b1p (check-wbuf0? (MA-LSU MA))))
```

```
                (not (b1p (check-wbuf1? (MA-LSU MA))))
                (not (b1p (release-rbuf? (MA-LSU MA) MA sigs))))
           (not (member-equal (INST-stg (step-INST i MT MA sigs))
                              '((LSU lch)
                                (LSU wbuf0 lch)
                                (LSU wbuf1 lch)))))
  :hints (("Goal" :in-theory (e/d (step-inst-low-level-functions
                                   step-inst-execute-inst)
                                  (inst-is-at-one-of-the-stages))
                :use (:instance inst-is-at-one-of-the-stages))))

(encapsulate nil
(local
(defthm no-inst-at-LSU-lch-MT-step-not-check-release-help
    (implies (and (inv MT MA)
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (check-wbuf0? (MA-LSU MA))))
                (not (b1p (check-wbuf1? (MA-LSU MA))))
                (not (b1p (release-rbuf? (MA-LSU MA) MA sigs))))
           (no-inst-at-stgs-in-trace '((LSU lch)
                                       (LSU wbuf0 lch)
                                       (LSU wbuf1 lch))
                                     (step-trace trace MT MA sigs
                                                 ISA spc smc)))
  :hints (("Goal" :in-theory (disable member-equal)))))

(defthm no-inst-at-LSU-lch-MT-step-not-check-release
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (check-wbuf0? (MA-LSU MA))))
                (not (b1p (check-wbuf1? (MA-LSU MA))))
                (not (b1p (release-rbuf? (MA-LSU MA) MA sigs))))
           (no-inst-at-stgs '((LSU lch) (LSU wbuf0 lch)
                              (LSU wbuf1 lch))
                            (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable no-inst-at-stgs))))
)

(defthm no-inst-at-LSU-lch-MT-step
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (not (b1p (LSU-latch-valid? (LSU-lch (step-LSU MA sigs))))))
           (no-inst-at-stgs '((LSU lch) (LSU wbuf0 lch)
                              (LSU wbuf1 lch))
                            (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable step-LSU step-LSU-lch lift-b-ops))))

(defthm no-LSU-stg-conflict-preserved
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (MA-input-p sigs)
                (not (MT-CMI-p (MT-step MT MA sigs))))
           (no-LSU-stg-conflict (MT-step MT MA sigs) (MA-step MA sigs)))
  :hints (("goal" :in-theory (enable no-LSU-stg-conflict))))

(defthm no-stage-conflict-preserved
    (implies (and (inv MT MA)
                (MAETT-p MT) (MA-state-p MA)
                (MA-input-p sigs)
                (not (MT-CMI-p (MT-step MT MA sigs))))
           (no-stage-conflict (MT-step MT MA sigs) (MA-step MA sigs)))
```

```
  :hints (("goal" :in-theory (enable no-stage-conflict))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Proof about no-tag-conflict
;;  Prove that each dispatched instruction has a unique Tomasulo's tag.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Proof of no-tag-conflict for initial states

(encapsulate nil
(local
(defthm not-robe-valid-if-MA-flushed-help
    (implies (and (b1p (robe-empty-under? idx2 ROB))
                  (integerp idx) (integerp idx2) (< idx idx2) (<= 0 idx))
             (not (b1p (robe-valid? (nth-robe idx ROB)))))
  :hints (("goal" :in-theory (enable lift-b-ops robe-empty-under?
                                     robe-empty?)))))

(defthm not-robe-valid-if-MA-flushed
    (implies (and (b1p (MA-flushed? MA)) (rob-index-p idx))
             (not (b1p (robe-valid? (nth-robe idx (MA-rob MA))))))
  :hints (("goal" :in-theory (enable MA-flushed? ROB-entries-empty? lift-b-ops
                                     rob-index-p unsigned-byte-p))))
)

(defthm no-tag-conflict-at-init-MT
    (implies (and (MA-state-p MA) (rob-index-p idx)
                  (b1p (MA-flushed? MA)))
             (no-tag-conflict-at idx (init-MT MA) MA))
  :hints (("goal" :in-theory (enable no-tag-conflict-at init-MT
                                     NO-inst-of-tag))))

(encapsulate nil
(local
(defthm no-tag-conflict-init-MT-help
    (implies (and (MA-state-p MA) (b1p (MA-flushed? MA))
                  (integerp index) (<= 0 index) (<= index *rob-size*))
             (no-tag-conflict-under index (init-MT MA) MA))
  :hints (("goal" :in-theory (enable rob-index-p UNSIGNED-BYTE-P)))))

(defthm no-tag-conflict-init-MT
    (implies (and (MA-state-p MA) (b1p (MA-flushed? MA)))
             (no-tag-conflict (init-MT MA) MA))
  :hints (("goal" :in-theory(enable no-tag-conflict))))
)

; Proof of no-tag-conflict-preserved
(defthm not-execute-stg-p-step-inst-if-not-DEO
    (implies (and (INST-p i)
                  (not (equal (INST-stg i) '(DQ 0)))
                  (not (execute-stg-p (INST-stg i))))
             (not (execute-stg-p (INST-stg (step-inst i MT MA sigs)))))
  :hints (("Goal" :in-theory (e/d (step-inst-low-level-functions
                                     dq-stg-p step-inst-dq-inst)
                                  (inst-is-at-one-of-the-stages))
                  :use (:instance inst-is-at-one-of-the-stages))))

(defthm not-execute-stg-p-step-inst-if-not-dispatch-inst
    (implies (and (INST-p i)
                  (not (b1p (dispatch-inst? MA)))
                  (not (execute-stg-p (INST-stg i))))
             (not (execute-stg-p (INST-stg (step-inst i MT MA sigs)))))
  :hints (("Goal" :in-theory (e/d (step-inst-low-level-functions
```

```
                                   dq-stg-p step-inst-dq-inst)
                                (inst-is-at-one-of-the-stages))
                  :use (:instance inst-is-at-one-of-the-stages))))

(defthm not-complete-stg-p-step-inst-if-not-DE0
    (implies (and (INST-p i)
                  (not (equal (INST-stg i) '(DQ 0)))
                  (not (execute-stg-p (INST-stg i)))
                  (not (complete-stg-p (INST-stg i))))
             (not (complete-stg-p (INST-stg (step-inst i MT MA sigs)))))
  :hints (("Goal" :in-theory (e/d (step-inst-low-level-functions
                                   dq-stg-p step-inst-dq-inst)
                                (inst-is-at-one-of-the-stages))
                  :use (:instance inst-is-at-one-of-the-stages))))

(defthm not-complete-stg-p-step-inst-if-not-dispatch-inst
    (implies (and (INST-p i)
                  (not (b1p (dispatch-inst? MA)))
                  (not (execute-stg-p (INST-stg i)))
                  (not (complete-stg-p (INST-stg i))))
             (not (complete-stg-p (INST-stg (step-inst i MT MA sigs)))))
  :hints (("Goal" :in-theory (e/d (step-inst-low-level-functions
                                   dq-stg-p step-inst-dq-inst)
                                (inst-is-at-one-of-the-stages))
                  :use (:instance inst-is-at-one-of-the-stages))))

(defthm complete-stg-p-step-inst-if-not-commit
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (or (not (equal (MT-ROB-head MT) (INST-tag i)))
                      (not (b1p (commit-inst? MA))))
                  (complete-stg-p (INST-stg i))
                  (INST-p i) (MAETT-p MT) (MA-state-p MA))
             (complete-stg-p (INST-stg (step-inst i MT MA sigs))))
  :hints (("Goal" :in-theory (enable step-inst-low-level-functions
                                     INST-commit? lift-b-ops
                                     step-inst-complete-inst))))

(defthm INST-tag-step-inst-opener
    (implies (and (INST-p i) (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (equal (INST-tag (step-INST i MT MA sigs))
                    (if (and (equal (INST-stg i) '(DQ 0))
                             (b1p (DISPATCH-INST? MA)))
                        (ROB-tail (MA-ROB MA))
                        (INST-tag i))))
  :hints (("goal" :in-theory (e/d (step-inst-low-level-functions
                                   step-inst-dq-inst dq-stg-p)
                                (inst-is-at-one-of-the-stages))
                  :use (:instance inst-is-at-one-of-the-stages))))

(defthm robe-valid-MT-ROB-tail-if-dispatch-inst
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (b1p (dispatch-inst? MA)))
             (equal (robe-valid? (nth-robe (MT-ROB-tail MT) (MA-rob MA)))
                    0))
  :hints (("Goal" :in-theory (enable dispatch-inst? lift-b-ops
                                     equal-b1p-converter
                                     dispatch-to-IU?
                                     dispatch-to-MU?
                                     dispatch-to-BU?
                                     dispatch-to-LSU?
```

```
                                    DQ-ready-no-unit?
                                    dispatch-no-unit?))))

(defthm equal-MT-ROB-head-MT-ROB-tail
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA))
             (iff (equal (MT-ROB-head MT) (MT-ROB-tail MT))
                  (or (b1p (ROB-full? (MA-ROB MA)))
                      (b1p (ROB-empty? (MA-ROB MA))))))
     :hints (("Goal" :in-theory (enable ROB-full? ROB-empty? lift-b-ops
                                        lift-b-ops))))

(encapsulate nil
(local
(defthm uniq-inst-of-tag-MT-step-if-robe-valid-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (rob-index-p idx)
                  (no-inst-of-tag-in-trace idx trace)
                  (or (not (equal (MT-ROB-head MT) idx))
                      (not (b1p (commit-inst? MA))))
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (robe-valid? (nth-robe idx (MA-rob MA)))))
             (no-inst-of-tag-in-trace idx (step-trace trace MT MA sigs
                                                      ISA spc smc)))
  :hints (("goal" :induct t
                  :in-theory (enable committed-p dispatched-p)))))

(local
(defthm uniq-inst-of-tag-MT-step-if-robe-valid-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (rob-index-p idx)
                  (uniq-inst-of-tag-in-trace idx trace)
                  (or (not (equal (MT-ROB-head MT) idx))
                      (not (b1p (commit-inst? MA))))
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (robe-valid? (nth-robe idx (MA-rob MA)))))
             (uniq-inst-of-tag-in-trace idx (step-trace trace MT MA sigs
                                                        ISA spc smc)))
  :hints (("goal" :induct t
                  :in-theory (enable committed-p dispatched-p)))))

(defthm uniq-inst-of-tag-MT-step-if-robe-valid
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (rob-index-p idx)
                  (or (not (equal (MT-ROB-head MT) idx))
                      (not (b1p (commit-inst? MA))))
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (robe-valid? (nth-robe idx (MA-rob MA)))))
             (uniq-inst-of-tag idx (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable uniq-inst-of-tag)
                  :use (:instance UNIQ-INST-OF-TAG-IF-ROBE-VALID))))

)

(defthm INST-at-DE0-be-dispatched-if-dispatch-inst
    (implies (and (inv MT MA)
                  (equal (INST-stg i) '(DQ 0))
```

```
                (b1p (dispatch-inst? MA))
                (MAETT-p MT) (MA-state-p MA)
                (not (execute-stg-p (INST-stg (step-inst i MT MA sigs)))))
           (complete-stg-p (INST-stg (step-inst i MT MA sigs))))
  :hints (("Goal" :in-theory (enable dispatch-inst? lift-b-ops))))

(encapsulate nil
(local
(defthm not-execute-stg-p-step-inst-if-robe-receive-inst
    (implies (and (inv MT MA)
                  (not (equal (INST-stg i) '(DQ 0)))
                  (b1p (robe-receive-inst? (MA-rob MA) (INST-tag i) MA))
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA))
             (not (execute-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("Goal" :in-theory (e/d (robe-receive-inst? lift-b-ops
                                   dispatch-inst? dispatch-no-unit?
                                   dispatch-to-IU? dispatch-to-MU?
                                   dispatch-to-BU? dispatch-to-LSU?)
                                  (inst-is-at-one-of-the-stages))
                  :use (:instance inst-is-at-one-of-the-stages)))))

(local
(defthm not-complete-stg-p-step-inst-if-robe-receive-inst
    (implies (and (inv MT MA)
                  (INST-in i MT) (INST-p i)
                  (MAETT-p MT) (MA-state-p MA)
                  (not (equal (INST-stg i) '(DQ 0)))
                  (b1p (robe-receive-inst? (MA-rob MA) (INST-tag i) MA)))
             (not (complete-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("Goal" :in-theory (e/d (robe-receive-inst? lift-b-ops
                                   dispatch-inst? dispatch-no-unit?
                                   dispatch-to-IU? dispatch-to-MU?
                                   dispatch-to-BU? dispatch-to-LSU?)
                                  (inst-is-at-one-of-the-stages))
                  :use (:instance inst-is-at-one-of-the-stages)))))

(local
(defthm uniq-inst-of-tag-MT-step-if-robe-receive-inst-help-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (rob-index-p idx)
                  (no-inst-at-stg-in-trace '(DQ 0) trace)
                  (b1p (robe-receive-inst? (MA-ROB MA) idx MA)))
             (no-inst-of-tag-in-trace idx
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))
  :hints (("Goal" :in-theory (enable robe-receive-inst? lift-b-ops
                                     committed-p dispatched-p)))))

(local
(defthm uniq-inst-of-tag-MT-step-if-robe-receive-inst-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (rob-index-p idx)
                  (not (b1p (flush-all? MA sigs)))
                  (uniq-inst-at-stg-in-trace '(DQ 0) trace)
                  (b1p (robe-receive-inst? (MA-ROB MA) idx MA)))
```

```
            (uniq-inst-of-tag-in-trace idx
                                       (step-trace trace MT MA sigs
                                                   ISA spc smc)))
  :hints (("Goal" :in-theory (enable robe-receive-inst? lift-b-ops
                                     committed-p dispatched-p)))))

(defthm uniq-inst-of-tag-MT-step-if-robe-receive-inst
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (MA-input-p sigs)
                  (rob-index-p idx)
                  (not (b1p (flush-all? MA sigs)))
                  (b1p (robe-receive-inst? (MA-ROB MA) idx MA)))
             (uniq-inst-of-tag idx (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable uniq-inst-of-tag uniq-inst-at-stg
                                     robe-receive-inst?
                                     lift-b-ops)
                  :use (:instance UNIQ-INST-AT-STG-IF-DQ-DEO-VALID))))
)

(defthm uniq-inst-of-tag-MT-step
    (implies (and (inv MT MA) (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (rob-index-p idx)
                  (b1p (robe-valid? (nth-robe idx (step-ROB MA sigs)))))
             (uniq-inst-of-tag idx (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable step-robe lift-b-ops))))

(defthm not-execute-stg-p-step-inst-if-INST-cause-jmp
    (implies (and (inv MT MA)
                  (INST-p i) (MAETT-p MT) (MA-state-p MA)
                  (b1p (INST-cause-jmp? i MT MA sigs)))
             (not (execute-stg-p (INST-stg (step-inst i MT MA sigs)))))
  :hints (("Goal" :use (:instance inst-is-at-one-of-the-stages)
                  :in-theory (disable inst-is-at-one-of-the-stages))))

(defthm not-complete-stg-p-step-inst-if-INST-cause-jmp
    (implies (and (inv MT MA)
                  (INST-p i) (MAETT-p MT) (MA-state-p MA)
                  (b1p (INST-cause-jmp? i MT MA sigs)))
             (not (complete-stg-p (INST-stg (step-inst i MT MA sigs)))))
  :hints (("Goal" :use (committed-p-step-inst-if-INST-cause-jmp)
                  :in-theory
                  (disable committed-p-step-inst-if-INST-cause-jmp))))

(encapsulate nil
(local
(defthm no-inst-of-tag-MT-step-if-flush-all-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MT-all-commit-before-trace trace MT)
                  (MAETT-p MT) (MA-state-p MA)
                  (rob-index-p idx)
                  (b1p (flush-all? MA sigs)))
             (no-inst-of-tag-in-trace idx
                                      (step-trace trace MT MA sigs ISA
                                                  spc smc)))
  :hints (("goal" :in-theory (enable COMMITTED-P dispatched-p))
          (when-found (MT-ALL-COMMIT-BEFORE-TRACE (CDR TRACE)
                                                  MT)
                      (:cases ((committed-p (car trace)))))
          (when-found (execute-stg-p (INST-stg (step-inst (car trace)
                                                          MT MA sigs)))
```

```
                    (:cases ((committed-p (car trace)))))
          (when-found (complete-stg-p (INST-stg (step-inst (car trace)
                                                           MT MA sigs)))
                    (:cases ((committed-p (car trace))))))))

(defthm no-inst-of-tag-MT-step-if-flush-all
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (rob-index-p idx)
                  (b1p (flush-all? MA sigs)))
             (no-inst-of-tag idx (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable no-inst-of-tag))))
)

(encapsulate nil
(local
(defthm no-inst-of-tag-MT-step-if-not-robe-receive-inst-help
    (implies (and (inv MT MA)
                  (subtrace-p trace MT) (INST-listp trace)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (rob-index-p idx)
                  (not (b1p (robe-receive-inst? (MA-rob MA) idx MA)))
                  (not (b1p (robe-valid? (nth-robe idx (MA-rob MA))))))
             (no-inst-of-tag-in-trace idx
                                      (step-trace trace MT MA sigs
                                                  ISA spc smc)))
  :hints (("goal" :in-theory (enable ROBE-RECEIVE-INST? lift-b-ops
                                     committed-p dispatched-p))
          (when-found (EXECUTE-STG-P (INST-STG (STEP-INST (CAR TRACE)
                                                          MT MA SIGS)))
                      (:cases ((execute-stg-p (INST-stg (car trace))))))
          (when-found (complete-STG-P (INST-STG (STEP-INST (CAR TRACE)
                                                           MT MA SIGS)))
                      (:cases ((execute-stg-p (INST-stg (car trace)))
                               (complete-stg-p (INST-stg (car trace))))))))))

(defthm no-inst-of-tag-MT-step-if-not-robe-receive-inst
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (rob-index-p idx)
                  (not (b1p (robe-receive-inst? (MA-rob MA) idx MA)))
                  (not (b1p (robe-valid? (nth-robe idx (MA-rob MA))))))
             (no-inst-of-tag idx (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable no-inst-of-tag))))
)

(defthm not-complete-stg-p-step-inst-if-commit-inst
    (implies (and (inv MT MA)
                  (INST-p i)
                  (b1p (commit-inst? MA))
                  (equal (INST-tag i) (MT-ROB-head MT))
                  (complete-stg-p (INST-stg i))
                  (MAETT-p MT) (MA-state-p MA))
             (not (complete-stg-p (INST-stg (step-INST i MT MA sigs)))))
  :hints (("Goal" :in-theory (enable step-inst-complete-inst
                                     inst-commit? lift-b-ops
                                     step-inst-low-level-functions))))

(encapsulate nil
(local
(defthm no-inst-at-MT-rob-head-MT-step-help
    (implies (and (inv MT MA)
```

```
                (subtrace-p trace MT) (INST-listp trace)
                (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                (b1p (commit-inst? MA)))
           (no-inst-of-tag-in-trace (MT-rob-head MT)
                                    (step-trace trace MT MA sigs
                                                ISA spc smc)))
    :hints (("goal" :in-theory (enable  lift-b-ops
                                        committed-p dispatched-p))
          (when-found (EXECUTE-STG-P (INST-STG (STEP-INST (CAR TRACE)
                                                          MT MA SIGS)))
                      (:cases ((execute-stg-p (INST-stg (car trace))))))
          (when-found (complete-STG-P (INST-STG (STEP-INST (CAR TRACE)
                                                           MT MA SIGS)))
                      (:cases ((execute-stg-p (INST-stg (car trace)))
                               (complete-stg-p (INST-stg (car trace))))))))))

(defthm no-inst-at-MT-rob-head-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (b1p (commit-inst? MA)))
             (no-inst-of-tag (MT-rob-head MT) (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable no-inst-of-tag))))
)

(defthm no-inst-of-tag-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (rob-index-p idx)
                  (not (b1p (robe-valid? (nth-robe idx (step-ROB MA sigs))))))
             (no-inst-of-tag idx (MT-step MT MA sigs)))
  :hints (("Goal" :in-theory (enable step-robe lift-b-ops))))

; This is the individual case.  The rest is done by induction.
(defthm no-tag-conflict-at-MT-step
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (rob-index-p idx))
             (no-tag-conflict-at idx (MT-step MT MA sigs) (MA-step MA sigs)))
  :hints (("Goal" :in-theory (enable no-tag-conflict-at))))

(encapsulate nil
(local
(defthm no-tag-conflict-preserved-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs)
                  (integerp n) (<= 0 n)
                  (<= n 8))
             (no-tag-conflict-under n (MT-step MT MA sigs)
                                    (MA-step MA sigs)))
  :hints (("Goal" :in-theory (enable rob-index-p UNSIGNED-BYTE-P)))))

(defthm no-tag-conflict-preserved
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA) (MA-input-p sigs))
             (no-tag-conflict (MT-step MT MA sigs) (MA-step MA sigs)))
  :hints (("Goal" :in-theory (enable NO-TAG-CONFLICT))))
)
```

D.6.10 invariant-proof.lisp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MI-inv.lisp
; Author  Jun Sawada, University of Texas at Austin
;
;  This book combines the proof of all properties and
;  actually proves that inv is an invariant.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(in-package "ACL2")

(include-book "MA2-lemmas")
(include-book "MAETT-lemmas")
(local (include-book "wk-inv"))
(local (include-book "MI-inv"))
(local (include-book "ISA-comp"))
(local (include-book "misc-inv"))
(local (include-book "uniq-inv"))
(local (include-book "in-order"))
(local (include-book "reg-ref"))
(deflabel begin-invariant-proof)

(defthm inv-initial-MT
    (implies (and (MA-state-p MA) (flushed-p MA))
             (inv (init-MT MA) MA))
  :hints (("goal" :expand (inv (init-MT MA) MA))))

(defthm inv-step
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (MA-input-p sigs)
                  (not (MT-CMI-p (MT-step MT MA sigs))))
             (inv (MT-step MT MA sigs)
                  (MA-step MA sigs)))
  :hints (("Goal" :expand (inv (MT-step MT MA sigs)
                               (MA-step MA sigs))))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (MAETT-p MT)
                           (MA-state-p MA)
                           (MA-input-p sigs)
                           (not (inv (MT-step MT MA sigs)
                                     (MA-step MA sigs))))
                      (MT-CMI-p (MT-step MT MA sigs))))))

;  Invariant proof is completed.  The reader can skip the following comment.

#|
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; An alternative definition of MT-step and its invariant proof
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  We have an alternative definition of MT-step, which is defined as
;  MT-step*.  MT-step and MT-stepn* are slightly different.
;
; From here, we try to modify the definition of
; INST-step and INST-init.  The original definition of MT-step and
; the functions used to define it are a little bit intended for the proof
; efficiency, and not for presentation.  Here we give a simpler definitions
; of INST-step and INST-init, hence, MT-step, which must be easier for the
```

```
; reader. This is proven for FMSD paper.
;
;  Inst-init returns the initial INST state.
(defun INST-init* (MT MA sigs)
  (if (b1p (fetch-inst? MA sigs))
      (fetched-inst MT (MT-final-ISA MT)
                    (MT-specultv? MT)
                    (MT-self-modify? MT))
      (if (b1p (ex-intr? MA sigs))
          (exintr-INST MT  (MT-final-ISA MT) (MT-self-modify? MT))
          nil)))

(defun trace-pre-modified? (i trace smc)
  (if (endp trace)
      smc
      (if (equal (car trace) i)
          smc
          (trace-pre-modified? i (cdr trace) (INST-modified? (car trace))))))

; MT-pre-modified? determines if any instruction preceding i are
; modified.
(defun MT-pre-modified? (i MT)
  (trace-pre-modified? i (MT-trace MT) 0))
(in-theory (disable MT-pre-modified?))

; INST-step updates INST.  In the original definition, we used
; both exintr-INST and INST-step to define the next state of INST.
(defun INST-step* (i MT MA sigs)
  (if (and (b1p (INST-exintr-now? i MA sigs))
           (not (b1p (INST-cause-jmp? i MT MA sigs))))
      (exintr-INST MT (INST-pre-ISA i) (MT-pre-modified? i MT))
      (step-INST i MT MA sigs)))

; This is a simplified version of step-trace.
(defun step-trace* (trace MT MA sigs)
    (if (endp trace)
        (if (or (b1p (fetch-inst? MA sigs))
                (b1p (ex-intr? MA sigs)))
            (list (INST-init* MT MA sigs))
            nil)
        (b-if (INST-cause-jmp? (car trace) MT MA sigs)
              (list (INST-step* (car trace) MT MA sigs))
              (b-if (INST-exintr-now? (car trace) MA sigs)
                    (list (INST-step* (car trace) MT MA sigs))
                    (cons (INST-step* (car trace) MT MA sigs)
                          (step-trace* (cdr trace) MT MA sigs))))))

; This is a simplified version MT-step.
(defun MT-step* (MT MA sigs)
  (MAETT (MT-init-ISA MT)
         (1+ (MT-new-ID MT))
         (step-MT-dq-len MT MA sigs)
         (step-MT-wb-len MT MA sigs)
         (step-MT-rob-flg MT MA sigs)
         (step-MT-rob-head MT MA sigs)
         (step-MT-rob-tail MT MA sigs)
         (step-trace* (MT-trace MT) MT MA sigs)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Here is a proof for an alternative definition of MT-step
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(local
```

```
(encapsulate nil
(local
(defthm INST-pre-ISA-car-ISA-before-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (consp sub)
                  (tail-p sub trace)
                  (INST-listp trace))
             (equal (INST-pre-ISA (car sub))
                    (ISA-at-tail sub trace (INST-pre-ISA (car trace)))))
  :hints ((when-found (INST-PRE-ISA (CAR (CDR TRACE)))
                      (:cases ((consp (cdr trace))))))
  :rule-classes nil))

(defthm INST-pre-ISA-car-ISA-before
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (consp trace)
                  (subtrace-p trace MT)
                  (INST-listp trace))
             (equal (ISA-before trace MT) (INST-pre-ISA (car trace))))
  :hints (("goal" :in-theory (enable subtrace-p ISA-before)
                  :use (:instance INST-pre-ISA-car-ISA-before-help
                                  (sub trace) (trace (MT-trace MT))))
          ("goal'" :cases ((consp (MT-trace MT))))))
))

(local
(encapsulate nil
(local
(defthm INST-modified-car-ISA-before-help
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (consp sub)
                  (tail-p sub trace)
                  (INST-listp trace))
             (equal (trace-pre-modified? (car sub) trace smc)
                    (modified-inst-before-tail sub trace smc)))
  :hints (("goal" :expand (TRACE-PRE-MODIFIED? (CAR SUB) TRACE SMC)))))

(defthm INST-modified-car-ISA-before
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (consp trace)
                  (subtrace-p trace MT)
                  (INST-listp trace))
             (equal (modified-inst-before? trace MT)
                    (MT-pre-modified? (car trace) MT)))
  :hints (("goal" :in-theory (enable modified-inst-before? subtrace-p
                                     MT-pre-modified?))))
))

(local
(defthm new-step-trace
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA)
                  (subtrace-p trace MT)
                  (INST-listp trace))
             (equal (step-trace trace MT MA sigs
                                (ISA-before trace MT)
```

```
                                (specultv-before? trace MT)
                                (modified-inst-before? trace MT))
                    (step-trace* trace MT MA sigs)))
  :rule-classes nil))


; This lemma shows that a simplified version of MT-step is equivalent to
; the original definition,
(defthm MT-step-MT-step*
    (implies (and (inv MT MA)
                  (MAETT-p MT) (MA-state-p MA))
             (equal (MT-step* MT MA sigs)
                    (MT-step MT MA sigs)))
  :Hints (("goal" :in-theory (enable MT-step)
                  :use (:instance new-step-trace
                                  (trace (MT-trace MT))))))

(defthm inv-step*
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (MA-input-p sigs)
                  (not (MT-CMI-p (MT-step MT MA sigs))))
             (inv (MT-step* MT MA sigs)
                  (MA-step MA sigs))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  End of the proof of the invariant for the alternative definition.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
|#
```

D.7 Correctness Proof

D.7.1 correctness.lisp

Proof of the correctness. Please see Chapter 6 for discussions.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; correctness.lisp:
; Author  Jun Sawada, University of Texas at Austin
;
;  This book provides the proof of our correctness criteria.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(in-package "ACL2")

(include-book "MAETT-lemmas")
(include-book "wk-inv")
(include-book "invariant-proof")

(deflabel begin-exintr)
; This file contains the high level proof of our correctness criterion.
;
; Index
;   Def of various predicates.
;    map-inputs
;    zero-intr-lst
;    exintr-free-p
```

```
;    MT-len MT-num-commit-insts
;    num-insts
;    MT-all-retired-p
;   Lemmas related to input signal mapping and number of instructions.
;   Lemmas about ISA-stepn-fetched-from and ISA-self-modify-p
;   Invariant Proofs
;     inv-initial-MT
;     inv-stepn
;    MA-flushed-implies-MT-all-retired-p and lemmas about MT-all-retired-p
;    flushed-MA-=-MT-final-ISA
;      with matching lemma of individual MA components
;    exintr-correctness-criteria
;    correctness-criteria

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Construction of Witness functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; We construct the ISA input sequence for the final theorem from the
; initial MA state and MA input sequence.  MT-exintr-lst
; extracts an input sequence from the MAETT, by collecting values at
; INST-exintr? field of each MAETT entry. Function map-inputs first
; constructs the MAETT for the final MA state and then extracts
; external interrupt signals from it.
(defun trace-exintr-lst (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace)
      nil
      (cons (ISA-input (INST-exintr? (car trace)))
            (trace-exintr-lst (cdr trace)))))

(defun MT-exintr-lst (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (trace-exintr-lst (MT-trace MT)))

(in-theory (disable MT-exintr-lst))

; The witness function for the corresponding ISA sequence.
(defun map-inputs (MA sigs-lst n)
  (declare (xargs :guard (and (MA-state-p MA) (MA-input-listp sigs-lst)
                              (integerp n) (<= 0 n))))
  (MT-exintr-lst (MT-stepn (init-MT MA) MA sigs-lst n)))

(in-theory (disable map-inputs))

; MT-exintr-lst constructs a list of external interrupt input signals
; for all instructions in the MAETT, while another function
; MT-inputs-for-committed extracts external input signal for only
; committed instructions in a MAETT.  Since instructions commit in
; order, inputs-for-committed returns an initiating sublist of the
; input sequence which would be returned by MT-exintr-lst.  When all
; the instructions in a MAETT are committed, inputs-for-committed and
; MT-exintr-lst return the same input list.
(defun trace-inputs-for-committed (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace)
      nil
      (if (or (commit-stg-p (INST-stg (car trace)))
              (retire-stg-p (INST-stg (car trace))))
          (cons (ISA-input (INST-exintr? (car trace)))
                (trace-inputs-for-committed (cdr trace)))
          nil)))
```

```
(defun MT-inputs-for-committed (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (trace-inputs-for-committed (MT-trace MT)))

(in-theory (disable MT-inputs-for-committed))

; Each ISA input contains a bit indicating whether an external
; interrupt occurs during the execution of a current instruction; bit
; 1 indicates that an external interrupt takes place in this cycle
; while 0 corresponds to normal execution is performed. zero-intr-lst
; returns a list of n 0's, which is the input signal sequence that
; makes the ISA run without external interrupts.
(defun zero-intr-lst (n)
  (declare (xargs :guard (and (integerp n) (<= 0 n))))
  (if (zp n) nil (cons (ISA-input 0) (zero-intr-lst (1- n)))))

; A bit-level predicate to check whether a list of MA inputs contains
; no external interrupts at the MA level.
(defun exintr-free-p (sigs-lst)
  (declare (xargs :guard (and (MA-input-listp sigs-lst))))
  (if (endp sigs-lst)
      T
      (and (b1p (b-not (MA-input-exintr (car sigs-lst))))
           (exintr-free-p (cdr sigs-lst)))))

(defun trace-num-commit-insts (trace)
  (declare (xargs :guard (INST-listp trace)))
  (if (endp trace)
      0
    (if (or (retire-stg-p (INST-stg (car trace)))
            (commit-stg-p (INST-stg (car trace))))
        (1+ (trace-num-commit-insts (cdr trace)))
      0)))

; MT-num-commit-insts returns the number of committed instruction in
; a MAETT.  More precisely, it is the number of consecutively committed
; instructions from the beginning of a MAETT.
;
; Note that MT-num-commit-insts and MT-len return the same number
; for a MAETT whose instructions are all retired.
(defun MT-num-commit-insts (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (trace-num-commit-insts (MT-trace MT)))

; MT-len returns the number of instructions in a MAETT
(defun MT-len (MT)
  (declare (xargs :guard (MAETT-p MT)))
  (len (MT-trace MT)))

(in-theory (disable MT-len))

; The witness function.
;
; Num-insts returns the number of instructions which are completely or
; partially executed by the MA during the n cycle execution from an
; initial state MA.  If the MA state after n clock cycles is flushed,
; num-insts returns the exact number of instructions that have been
; completely executed.
;
; It first constructs the MAETT for the MA state after n clock cycles,
; and then count the instructions in the MAETT.
(defun num-insts (MA sigs-lst n)
```

```
  (declare (xargs :guard (and (MA-state-p MA) (MA-input-listp sigs-lst)
                              (integerp n) (<= 0 n)
                              (<= n (len sigs-lst)))
                  :measure (nfix n)))
  (MT-len (MT-stepn (init-MT MA) MA sigs-lst n)))

(in-theory (disable MT-num-commit-insts num-insts))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Following are lemmas about the number of instructions in a MAETT
;;; and input sequence  extracted from a MAETT.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(encapsulate nil
(local
 (defthm len-MT-exintr-lst-help
     (equal (len (trace-exintr-lst trace))
            (len trace))))

; The length of the input sequence returned by MT-exintr-lst.
(defthm len-MT-exintr-lst
    (equal (len (MT-exintr-lst MT)) (MT-len MT))
  :hints (("goal" :do-not-induct t
                  :in-theory (enable MT-exintr-lst MT-len))))

)

(encapsulate nil
(local
(defthm len-trace-inputs-for-committed
    (equal (len (trace-inputs-for-committed trace))
           (trace-num-commit-insts trace))))

; The length of the input sequence returned by MT-inputs-for-committed.
(defthm len-MT-inputs-for-committed
    (equal (len (MT-inputs-for-committed MT)) (MT-num-commit-insts MT))
  :hints (("Goal" :in-theory
                  (enable MT-inputs-for-committed MT-num-commit-insts))))
)

(defthm MT-exintr-lst-init-MT
    (equal (MT-exintr-lst (init-MT MA)) nil)
  :hints (("Goal" :in-theory (enable MT-exintr-lst))))

; The number of instructions at the initial MAETT is 0.
(defthm MT-len-init-MT
    (equal (MT-len (init-MT MA)) 0)
  :hints (("Goal" :in-theory (enable MT-len))))

; The number of instructions recorded in a MAETT may decrease as
; partially executed instructions can be abandoned.  However,
; committed instructions are never eliminated from a MAETT, so the
; number of committed instructions in a MAETT never decreases.
(encapsulate nil
(local
 (defthm MT-num-commit-insts-MT-step->=-MT-num-commit-insts-help
     (<= (trace-num-commit-insts trace)
         (trace-num-commit-insts (step-trace trace MT MA sigs ISA spc smc)))))

; Monotonicity of the committed instructions.
(defthm MT-num-commit-insts-MT-step->=-MT-num-commit-insts
    (<= (MT-num-commit-insts MT)
        (MT-num-commit-insts (MT-step MT MA sigs)))
```

```
  :hints (("goal" :in-theory (enable MT-step MT-num-commit-insts)
                  :do-not-induct t))
  :rule-classes :linear)
)

(defthm MT-num-commit-insts-MT-stepn->=-MT-num-commit-insts
    (<= (MT-num-commit-insts MT)
        (MT-num-commit-insts (MT-stepn MT MA sigs-lst n)))
  :rule-classes :linear)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Properties about ISA-stepn-fetches-from and
;;;; ISA-self-modify-p.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(local
 (defun induction-scheme1 (ISA ol1 ol2 n1 n2)
   (if (or (zp n1) (zp n2)) (list ISA ol1 ol2 n1 n2)
       (induction-scheme1 (ISA-step ISA (car ol1))
                          (cdr ol1) (cdr ol2) (1- n1) (1- n2)))))

; Suppose n1 <= n2.  If the ISA fetches an instruction from a certain
; address, addr, during the first n1 steps, it certainly fetches the
; same instruction in the first n2 steps.
(defthm ISA-stepn-fetches-from-head-p
    (implies (and (head-p sl1 sl2) (<= n1 n2)
                  (<= n1 (len sl1)) (<= n2 (len sl2))
                  (integerp n1) (integerp n2)
                  (ISA-stepn-fetches-from addr ISA sl1 n1))
             (ISA-stepn-fetches-from addr ISA sl2 n2))
  :hints (("goal" :induct (induction-scheme1 ISA sl1 sl2 n1 n2))))

; N step version.
(defthm ISA-self-modify-p-head-p
    (implies (and (head-p sl1 sl2) (<= n1 n2)
                  (integerp n1) (integerp n2)
                  (<= n1 (len sl1)) (<= n2 (len sl2))
                  (ISA-self-modify-p ISA sl1 n1))
             (ISA-self-modify-p ISA sl2 n2))
  :hints (("goal" :induct (induction-scheme1 ISA sl1 sl2 n1 n2)
                  :restrict ((ISA-stepn-fetches-from-head-p
                              ((sl1 (cdr sl1)) (n1 (1- n1)))))))))

(encapsulate nil
(local
(defthm head-p-MT-inputs-for-committed-MT-step-help
    (head-p (trace-inputs-for-committed trace)
            (trace-inputs-for-committed (step-trace trace
                                                    MT MA sigs ISA spc smc)))))

; The input sequence for the committed instructions is extended when
; the MAETT is updated.  Head-p is a predicate to check if a list is
; an initiating sublist.
(defthm head-p-MT-inputs-for-committed-MT-step
    (head-p (MT-inputs-for-committed MT)
            (MT-inputs-for-committed (MT-step MT MA sigs)))
  :hints (("goal" :in-theory (e/d (MT-inputs-for-committed MT-step)
                                  (head-p trace-inputs-for-committed))
                  :do-not-induct t)))
)

; If a MAETT contains a self-modifying sequence of committed
; instructions, the updated MAETT also contains a self-modifying
```

```
; sequence of committed instructions.  This is because committed
; instructions will never undo a commit, and the sequence of committed
; instructions in an MAETT only grows, but never shrinks.
(defthm ISA-self-modify-p-MT-step
    (implies (and (MAETT-p MT)
                  (MA-state-p MA)
                  (ISA-self-modify-p ISA
                                     (MT-inputs-for-committed MT)
                                     (MT-num-commit-insts MT)))
             (ISA-self-modify-p ISA
                                (MT-inputs-for-committed (MT-step MT MA sigs))
                                (MT-num-commit-insts (MT-step MT MA sigs))))
  :hints (("Goal" :restrict ((ISA-self-modify-p-head-p
                              ((sl1  (MT-inputs-for-committed MT))
                               (n1  (MT-num-commit-insts MT))))))))

; N-step version of the previous theorem.
(defthm ISA-self-modify-p-MT-stepn
    (implies (and (MAETT-p MT)
                  (MA-state-p MA)
                  (MA-input-listp sigs-lst)
                  (ISA-self-modify-p ISA
                                     (MT-inputs-for-committed MT)
                                     (MT-num-commit-insts MT)))
             (ISA-self-modify-p ISA
                        (MT-inputs-for-committed (MT-stepn MT MA sigs-lst n))
                        (MT-num-commit-insts (MT-stepn MT MA sigs-lst n)))))

(encapsulate nil
(local
(defthm ISA-stepn-fetches-INST-if-INST-is-committed
    (implies (and (weak-inv MT)
                  (MAETT-p MT)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (member-equal i trace)
                  (INST-p i)
                  (trace-all-commit-before i trace)
                  (or (commit-stg-p (INST-stg i))
                      (retire-stg-p (INST-stg i)))
                  (not (b1p (INST-exintr? i)))
                  (equal (ISA-pc (INST-pre-ISA i)) addr))
             (ISA-stepn-fetches-from addr
                                     (INST-pre-ISA (car trace))
                                     (trace-inputs-for-committed trace)
                                     (trace-num-commit-insts trace)))
  :rule-classes
  ((:rewrite :corollary
             (implies (and (weak-inv MT)
                           (MAETT-p MT)
                           (member-equal i trace)
                           (INST-listp trace) (INST-p i)
                           (subtrace-p trace MT)
                           (trace-all-commit-before i trace)
                           (or (commit-stg-p (INST-stg i))
                               (retire-stg-p (INST-stg i)))
                           (not (b1p (INST-exintr? i)))
                           (equal (INST-pre-ISA (car trace)) ISA)
                           (equal (ISA-pc (INST-pre-ISA i)) addr))
                      (ISA-stepn-fetches-from addr
                                              ISA
                                              (trace-inputs-for-committed trace)
```

```
                                        (trace-num-commit-insts trace)))))
  :hints (("goal" :induct t
                  :in-theory (enable ISA-FETCHES-FROM))
          (when-found (ISA-stepn-FETCHES-FROM ADDR
                                              (INST-PRE-ISA (CAR (CDR TRACE)))
               (TRACE-INPUTS-FOR-COMMITTED (CDR TRACE))
               (TRACE-NUM-COMMIT-INSTS (CDR TRACE)))
                      (:cases ((consp (cdr trace))))))))

(local
(defthm ISA-self-modify-p-if-MT-modify-p-help
    (implies (and (weak-inv MT)
                  (subtrace-p trace MT)
                  (MAETT-p MT)
                  (consp trace)
                  (member-equal i trace)
                  (INST-p i)
                  (INST-listp trace)
                  (or (commit-stg-p (INST-stg i))
                      (retire-stg-p (INST-stg i)))
                  (trace-all-commit-before i trace)
                  (trace-modify-p i trace))
             (ISA-self-modify-p (INST-pre-ISA (car trace))
                                (trace-inputs-for-committed trace)
                                (trace-num-commit-insts trace)))
  :hints (("goal" :induct t
                  :in-theory (e/d (INST-MODIFY-P)
                                  (ISA-chained-trace-p)))
          (when-found (ISA-self-modify-P
                       (INST-PRE-ISA (CAR (CDR TRACE)))
                       (TRACE-INPUTS-FOR-COMMITTED (CDR TRACE))
                       (TRACE-NUM-COMMIT-INSTS (CDR TRACE)))
                      (:cases ((consp (cdr trace))))))
  :rule-classes nil))

(local
(defthm ISA-self-modify-p-if-MT-modify-p
    (implies (and (weak-inv MT)
                  (MAETT-p MT)
                  (MT-modify-p i MT)
                  (INST-in i MT)
                  (INST-p i)
                  (or (commit-stg-p (INST-stg i))
                      (retire-stg-p (INST-stg i)))
                  (MT-all-commit-before i MT))
             (ISA-self-modify-p (MT-init-ISA MT)
                                (MT-inputs-for-committed MT)
                                (MT-num-commit-insts MT)))
  :hints (("Goal" :in-theory (enable INST-in MT-modify-p
                                     MT-inputs-for-committed
                                     MT-all-commit-before
                                     MT-num-commit-insts)
                  :cases ((consp (MT-trace MT)))
                  :do-not-induct t)
          (when-found-multiple
           ((CONSP (MT-TRACE MT))
            (ISA-SELF-MODIFY-P (MT-INIT-ISA MT)
                               (TRACE-INPUTS-FOR-COMMITTED (MT-TRACE MT))
                               (TRACE-NUM-COMMIT-INSTS (MT-TRACE MT))))
           (:use
            (:instance ISA-self-modify-p-if-MT-modify-p-help
                       (trace (MT-trace MT)))))))
```

```
) ; local

(local
(encapsulate nil
(local
 (defthm commit-stg-MI1-if-all-commit-before-MI2-help
    (implies (and (distinct-member-p trace)
                  (member-in-order MI1 MI2 trace)
                  (trace-all-commit-before MI2 trace)
                  (not (commit-stg-p (INST-stg MI1))))
             (retire-stg-p (INST-stg MI1)))
   :hints (("Goal" :in-theory (enable member-in-order*)))))

(defthm commit-stg-MI1-if-all-commit-before-MI2
    (implies (and (weak-inv MT)
                  (INST-in-order-p MI1 MI2 MT)
                  (MT-all-commit-before MI2 MT)
                  (not (commit-stg-p (INST-stg MI1))))
             (retire-stg-p (INST-stg MI1)))
  :Hints (("Goal" :in-theory (enable INST-in-order-p MT-all-commit-before
                                     weak-inv MT-distinct-INST-p)))
  :rule-classes nil)
))

(local
(encapsulate nil
(local
(defthm all-commit-before-MI1-if-all-commit-before-MI2-help
     (implies (and (distinct-member-p trace)
                   (member-in-order MI1 MI2 trace)
                   (trace-all-commit-before MI2 trace))
              (trace-all-commit-before MI1 trace))
  :hints (("Goal" :in-theory (enable member-in-order*)))))

(defthm all-commit-before-MI1-if-all-commit-before-MI2
     (implies (and (weak-inv MT)
                   (INST-in-order-p MI1 MI2 MT)
                   (MT-all-commit-before MI2 MT))
              (MT-all-commit-before MI1 MT))
  :hints (("Goal" :in-theory (enable MT-all-commit-before INST-in-order-p
                                     weak-inv MT-distinct-INST-p)))
   :rule-classes nil)
))

(local
(defthm ISA-self-modify-p-if-INST-modified-help
    (implies (and (weak-inv MT)
                  (MAETT-p MT)
                  (trace-correct-modified-flgs-p trace MT 0)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (member-equal i trace)
                  (INST-p i)
                  (MT-all-commit-before i MT)
                  (or (commit-stg-p (INST-stg i))
                      (retire-stg-p (INST-stg i)))
                  (b1p (INST-modified? i)))
             (ISA-self-modify-p (MT-init-ISA MT)
                                (MT-inputs-for-committed MT)
                                (MT-num-commit-insts MT)))
  :hints (("Goal" :induct t)
          (when-found (TRACE-CORRECT-MODIFIED-FLGS-P (CDR TRACE) MT '1)
```

```
             (:use ((:instance commit-stg-MI1-if-all-commit-before-MI2
                               (MI1 (car trace)) (MI2 i))
                    (:instance all-commit-before-MI1-if-all-commit-before-MI2
                               (MI1 (car trace)) (MI2 i))))))
  :rule-classes nil))


(local
 (defthm ISA-self-modify-p-if-INST-modified
     (implies (and (weak-inv MT)
                   (MAETT-p MT)
                   (INST-in i MT)
                   (INST-p i)
                   (or (commit-stg-p (INST-stg i))
                       (retire-stg-p (INST-stg i)))
                   (MT-all-commit-before i MT)
                   (b1p (INST-modified? i)))
              (ISA-self-modify-p (MT-init-ISA MT)
                                 (MT-inputs-for-committed MT)
                                 (MT-num-commit-insts MT)))
   :Hints (("Goal" :in-theory (enable weak-inv
                                      CORRECT-modified-FLGS-P
                                      INST-in)
           :use (:instance ISA-self-modify-p-if-INST-modified-help
                           (trace (MT-trace MT)))
           :do-not-induct t))))

(local
(defthm trace-all-commit-before-cadr-help
    (implies (and (distinct-member-p trace2)
                  (tail-p trace trace2)
                  (trace-all-commit-before (car trace) trace2)
                  (consp trace)
                  (or (commit-stg-p (INST-stg (car trace)))
                      (retire-stg-p (INST-stg (car trace)))))
             (trace-all-commit-before (cadr trace) trace2))))

(local
(defthm trace-all-commit-before-cadr
    (implies (and (weak-inv MT)
                  (MAETT-p MT)
                  (subtrace-p trace MT)
                  (consp trace)
                  (trace-all-commit-before (car trace) (MT-trace MT))
                  (or (commit-stg-p (INST-stg (car trace)))
                      (retire-stg-p (INST-stg (car trace)))))
             (trace-all-commit-before (cadr trace) (MT-trace MT)))
  :hints (("Goal" :in-theory (enable weak-inv MT-distinct-INST-p
                                     subtrace-p))))
)

(local
(defthm ISA-self-modify-p-if-MT-CMI-help
    (implies (and (weak-inv MT)
                  (MAETT-p MT)
                  (subtrace-p trace MT)
                  (INST-listp trace)
                  (implies (consp trace) (MT-all-commit-before (car trace) MT))
                  (not (ISA-self-modify-p (MT-init-ISA MT)
                                          (MT-inputs-for-committed MT)
                                          (MT-num-commit-insts MT))))
             (not (trace-CMI-p trace)))
```

```
  :hints (("goal" :in-theory (enable MT-CMI-p
                                     committed-p
                                     MT-all-commit-before)
           :restrict ((ISA-self-modify-p-if-INST-modified
                       ((i (car trace)))))
           :induct t))))

; MT-CMI-p and ISA-self-modify-p present the same concept of
; self-modification in different forms.  MT-CMI-p determines whether a
; MAETT records a self-modifying program.  ISA-self-modify-p
; determines from the initial ISA state whether the ISA executes
; self-modifying instructions in a certain number of steps.  The lemma
; states that, whenever MT-CMI-p is true suggesting the MAETT contains
; a self-modifying program, the ISA executes self-modifying program.
(defthm ISA-self-modify-p-if-MT-CMI
    (implies (and (weak-inv MT)
                  (MAETT-p MT)
                  (not (ISA-self-modify-p (MT-init-ISA MT)
                                          (MT-inputs-for-committed MT)
                                          (MT-num-commit-insts MT))))
             (not (MT-CMI-p MT)))
  :hints (("goal" :in-theory (enable MT-CMI-p
                                     MT-all-commit-before))))
)

; An initial MAETT does not contain any committed instructions.
(defthm not-MT-CMI-p-init-MT
    (implies (MA-state-p MA)
             (not (MT-CMI-p (init-MT MA))))
  :hints (("goal" :in-theory (enable init-MT MT-CMI-p))))

(local
 (defun induction-scheme3 (ISA trace)
   (if (endp trace)
       (list ISA trace)
       (induction-scheme3 (ISA-step ISA (ISA-input (INST-exintr? (car trace))))
                          (cdr trace)))))

(local
(defthm ISA-stepn-fetches-from-trace-num-commit-insts
    (implies (not (ISA-stepn-fetches-from addr ISA
                                          (trace-exintr-lst trace)
                                          (len trace)))
             (not (ISA-stepn-fetches-from addr ISA
                                          (trace-inputs-for-committed trace)
                                          (trace-num-commit-insts trace))))
  :hints (("Goal" :induct (induction-scheme3 ISA trace)))))

(local
(defthm ISA-self-modify-p-trace-num-commit-insts
    (implies (not (ISA-self-modify-p ISA (trace-exintr-lst trace)
                                     (len trace)))
             (not (ISA-self-modify-p ISA
                                     (trace-inputs-for-committed trace)
                                     (trace-num-commit-insts trace))))
  :hints (("Goal" :induct (induction-scheme3 ISA trace)))))

; Since (MT-len MT) >= (MT-num-commit-insts MT),
; the self-modification of the program that occurs during the first
; (MT-num-commit-insts MT) cycles of ISA execution occurs
; in the first (MT-len MT) cycles of execution.
```

```
(defthm ISA-self-modify-p-MT-inputs-for-committed
    (implies (not (ISA-self-modify-p ISA (MT-exintr-lst MT)
                                     (MT-len MT)))
             (not (ISA-self-modify-p ISA (MT-inputs-for-committed MT)
                                     (MT-num-commit-insts MT))))
  :hints (("Goal" :in-theory (enable MT-inputs-for-committed
                                     MT-num-commit-insts
                                     MT-exintr-lst
                                     MT-len))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Invariant Proofs
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
#|
; These lemmas are proved in invariant-proof.lisp
; We need two major lemmas to prove our correctness criterion.

; Initial execution trace satisfies invariants.
(defthm inv-initial-MT
    (implies (and (MA-state-p MA) (flushed-p MA))
             (inv (init-MT MA) MA)))

; The invariants are preserved as far as no modified
; instructions are committed.  This proof takes an extensive
; verification analysis.
(defthm inv-step
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (MA-input-p sigs)
                  (not (MT-CMI-p (MT-step MT MA sigs))))
             (inv (MT-step MT MA sigs)
                        (MA-step MA sigs)))
  :hints (("Goal" :in-theory (enable inv)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (MAETT-p MT)
                           (MA-state-p MA)
                           (MA-input-p sigs)
                           (not (inv (MT-step MT MA sigs)
                                     (MA-step MA sigs))))
                      (MT-CMI-p (MT-step MT MA sigs))))))

|#

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; stepn version of invariant lemmas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Stepn version of weak-inv lemma.
(defthm weak-inv-stepn
    (implies (and (weak-inv MT)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (MA-input-listp sigs-lst)
                  (<= n (len sigs-lst)))
             (weak-inv (MT-stepn MT MA sigs-lst n)))
  :hints (("goal" :in-theory (enable MA-stepn))))

; Stepn version of strong invariant lemma.
; Strong invariants are preserved during n cycles of MA execution as
```

```
; far as no modified instructions are committed during the execution.
(defthm inv-stepn
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (MA-input-listp sigs-lst)
                  (<= n (len sigs-lst))
                  (not (MT-CMI-p (MT-stepn MT MA sigs-lst n))))
             (inv (MT-stepn MT MA sigs-lst n)
                  (MA-stepn MA sigs-lst n)))
  :hints (("goal" :in-theory (enable MA-stepn)))
  :rule-classes
  ((:rewrite)
   (:rewrite :corollary
             (implies (and (inv MT MA)
                           (MAETT-p MT)
                           (MA-state-p MA)
                           (MA-input-listp sigs-lst)
                           (<= n (len sigs-lst))
                           (not (inv (MT-stepn MT MA sigs-lst n)
                                     (MA-stepn MA sigs-lst n))))
                      (MT-CMI-p (MT-stepn MT MA sigs-lst n))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  MA-flushed-implies-MT-all-retired-p and other lemmas with MT-all-retied
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(encapsulate nil
(local
(defthm trace-all-retired-implies-not-trace-specultv
    (implies (and (INST-listp trace)
                  (trace-no-specultv-commit-p trace)
                  (trace-all-retired trace))
             (not (b1p (trace-specultv? trace))))
  :hints (("goal" :in-theory (enable invariants-def lift-b-ops)))))

;; An MAETT whose instructions are all retired does not
;; contain any speculative instruction.
(defthm MT-all-retired-p-implies-not-MT-specultv-help
    (implies (and (MAETT-p MT)
                  (no-specultv-commit-p MT)
                  (MT-all-retired-p MT))
             (not (MT-specultv-p MT)))
  :hints (("Goal"  :in-theory (enable invariants-def MT-specultv?
                                      MT-specultv-p MT-specultv?
                                      MT-all-retired-p))))

(defthm MT-all-retired-p-implies-not-MT-specultv
    (implies (and (MAETT-p MT)
                  (inv MT MA)
                  (MT-all-retired-p MT))
             (not (MT-specultv-p MT)))
  :hints (("Goal"  :in-theory (enable inv))))
)

(encapsulate  nil
(local
(defthm MT-all-retired-p-implies-not-MT-self-modify-help
    (implies (and (not (trace-CMI-p trace))
                  (trace-all-retired trace))
             (equal (trace-self-modify? trace) 0))))

(defthm MT-all-retired-p-implies-not-MT-self-modify
```

```
    (implies (and (not (MT-CMI-p MT))
                  (MT-all-retired-p MT))
             (not (MT-self-modify-p MT)))
  :hints (("Goal" :in-theory (enable MT-all-retired-p MT-self-modify?
                                     MT-self-modify-p
                                     MT-self-modify?
                                     MT-CMI-p))))
)

(encapsulate nil
(local
(defthm MT-CMI-p-implies-MT-self-modify-help
    (implies (trace-CMI-p trace)
             (equal (trace-self-modify? trace) 1))))

(defthm MT-CMI-p-implies-MT-self-modify
    (implies (MT-CMI-p MT)
             (MT-self-modify-p MT))
  :hints (("goal" :in-theory (enable MT-CMI-p
                                     MT-self-modify-p
                                     MT-self-modify?))))
)

; If an MA state is flushed, the corresponding MAETT contains only
; retired instructions.
;
; Note: consistent-MA-p is one of the conjuncts used in the
; definition of inv.
(defthm MA-flushed-implies-INST-retired
    (implies (and (INST-p i)
                  (MA-state-p MA)
                  (flushed-p MA)
                  (INST-inv i MA)
                  (consistent-MA-p MA))
             (retire-stg-p (INST-stg i)))
  :hints (("goal" :in-theory (enable inst-inv-def
                                     MA-flushed-def lift-b-ops
                                     consistent-rob-p-forward)))
  :rule-classes
  ((:rewrite :corollary (implies (and (INST-p i)
                                      (MA-state-p MA)
                                      (flushed-p MA)
                                      (not (retire-stg-p (INST-stg i)))
                                      (consistent-MA-p MA))
                                 (not (INST-inv i MA))))))

;; Using the previous lemma, we can prove that all the instructions in an
;; MAETT are retired if the corresponding MA is flushed.
(encapsulate nil
(local
(defthm MA-flushed-implies-trace-all-retired
    (implies (and (trace-INST-inv trace MA)
                  (consistent-MA-p MA)
                  (INST-listp trace)
                  (MA-state-p MA)
                  (flushed-p MA))
             (trace-all-retired trace))))

(defthm MA-flushed-implies-MT-all-retired-p
    (implies (and (inv MT MA)
                  (MAETT-p MT)
                  (MA-state-p MA)
```

```
                (flushed-p MA))
           (MT-all-retired-p MT))
  :hints (("goal" :In-theory (enable inv weak-inv
                                     MT-INST-inv
                                     MT-all-retired-p))))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Proof of flushed-MA-=-MT-final-ISA
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; In this section, we prove flushed-MA-=-MT-final-ISA, which shows
;; that a final MA state is equivalent to the ISA state after executing
;; m instructions, where m = (MT-len MT).
;
;; We prove the lemma by showing that the states of individual
;; programmer visible components are the same in the MA and ISA.
;
(defthm ISA-pc-ISA-stepn-MT-pc-help
    (implies (ISA-chained-trace-p trace ISA)
             (equal (ISA-pc (ISA-stepn ISA
                                       (trace-exintr-lst trace)
                                       (len trace)))
                    (trace-pc trace (ISA-pc ISA))))
  :hints (("goal" :in-theory (enable ISA-stepn))))

; This theorem shows that the pc in ISA_m is correctly represented
; by (MT-pc MT) if all instructions are retired.
(defthm ISA-pc-ISA-stepn-MT-pc
    (implies (weak-inv MT)
             (equal (ISA-pc (ISA-stepn (MT-init-ISA MT)
                                       (MT-exintr-lst MT)
                                       (MT-len MT)))
                    (MT-pc MT)))
  :hints (("goal" :in-theory (enable MT-pc MT-exintr-lst MT-len
                                     weak-inv ISA-step-chain-p
                                     MT-all-retired-p))))

(defthm ISA-RF-ISA-stepn-MT-RF-help
    (implies (and (ISA-chained-trace-p trace ISA)
                  (trace-all-retired trace))
             (equal (ISA-RF (ISA-stepn ISA
                                       (trace-exintr-lst trace)
                                       (len trace)))
                    (trace-RF trace (ISA-RF ISA))))
  :hints (("goal" :in-theory (enable ISA-stepn))))

; This theorem shows that the RF in ISA_m is correctly represented
; by (MT-RF MT) if all instructions are retired.
(defthm ISA-RF-ISA-stepn-MT-RF
    (implies (and (weak-inv MT)
                  (MT-all-retired-p MT))
             (equal (ISA-RF (ISA-stepn (MT-init-ISA MT)
                                       (MT-exintr-lst MT)
                                       (MT-len MT)))
                    (MT-RF MT)))
  :hints (("goal" :in-theory (enable MT-RF MT-exintr-lst MT-len
                                     weak-inv ISA-step-chain-p
                                     MT-all-retired-p))))

(defthm ISA-SRF-ISA-stepn-MT-SRF-help
    (implies (and (ISA-chained-trace-p trace ISA)
                  (trace-all-retired trace))
```

```
            (equal (ISA-SRF (ISA-stepn ISA
                                       (trace-exintr-lst trace)
                                       (len trace)))
                   (trace-SRF trace (ISA-SRF ISA))))
  :hints (("goal" :in-theory (enable ISA-stepn))))

; This theorem shows that the SRF in ISA_m is correctly represented
; by (MT-SRF MT) if all instructions are retired.
(defthm ISA-SRF-ISA-stepn-MT-SRF
    (implies (and (weak-inv MT)
                  (MT-all-retired-p MT))
             (equal (ISA-SRF (ISA-stepn (MT-init-ISA MT)
                                        (MT-exintr-lst MT)
                                        (MT-len MT)))
                    (MT-SRF MT)))
  :hints (("goal" :in-theory (enable MT-SRF MT-exintr-lst MT-len
                                     weak-inv ISA-step-chain-p
                                     MT-all-retired-p))))

(defthm ISA-mem-ISA-stepn-MT-mem-help
    (implies (and (ISA-chained-trace-p trace ISA)
                  (trace-all-retired trace))
             (equal (ISA-mem (ISA-stepn ISA
                                        (trace-exintr-lst trace)
                                        (len trace)))
                    (trace-mem trace (ISA-mem ISA))))
  :hints (("goal" :in-theory (enable ISA-stepn))))

; This theorem shows that the memory in ISA_m is correctly represented
; by (MT-mem MT) if all instructions are retired.
(defthm ISA-mem-ISA-stepn-MT-mem
    (implies (and (weak-inv MT)
                  (MT-all-retired-p MT))
             (equal (ISA-mem (ISA-stepn (MT-init-ISA MT)
                                        (MT-exintr-lst MT)
                                        (MT-len MT)))
                    (MT-mem MT)))
  :hints (("goal" :in-theory (enable MT-mem MT-exintr-lst MT-len
                                     weak-inv ISA-step-chain-p
                                     MT-all-retired-p))))

(encapsulate nil
(local
(defthm MT-pc-MA-pc
    (implies (and (inv MT MA)
                  (not (MT-specultv-p MT))
                  (not (MT-self-modify-p MT)))
             (equal (MT-pc MT) (MA-pc MA)))
  :hints (("goal" :in-theory (enable inv pc-match-p)))))

(local
(defthm MT-RF-MA-RF
    (implies (inv MT MA)
             (equal (MT-RF MT) (MA-RF MA)))
  :hints (("goal" :in-theory (enable inv RF-match-p)))))

(local
(defthm MT-SRF-MA-SRF
    (implies (inv MT MA)
             (equal (MT-SRF MT) (MA-SRF MA)))
  :hints (("goal" :in-theory (enable inv SRF-match-p)))))
```

```
(local
(defthm MT-mem-MA-mem
    (implies (inv MT MA)
             (equal (MT-mem MT) (MA-mem MA)))
  :hints (("goal" :in-theory (enable inv mem-match-p)))))


;; Suppose the final state MA is flushed, and MA and its MAETT MT
;; satisfies the invariants.  Furthermore, assume there is no
;; committed instruction in MT which is self-modified by the program.
;; Then the projection of MA is equal to the post-ISA of the last
;; instruction in MT.  In other words, state MA is equivalent to the
;; final ISA state after executing the last instruction in MT

;; The right hand side represents the post-ISA of the last instruction.
;; Instead of representing the state as:
;; (post-ISA (car (last (MT-trace MT)))),
;; we expressed it as in the lemma.  The idea is as follows.  In a
;; well-formed MAETT, pre-ISA and post-ISA builds a chain of ISA state
;; transitions, as specified by a weak invariant ISA-step-chain-p.
;; If I_i and I_j are instructions in a well-formed MAETT, and I_i
;; immediately follows I_j.  Then
;; (pre-ISA I_j) = (post-ISA I_i) = (ISA-step (pre-ISA I_i) (INST-exintr? I_i))
;; Hence we can calculate the post-ISA of n'th entry of a well-formed
;; MAETT by  (ISA-stepn (pre-ISA (car MT)) intr-list n),  where
;; intr-list = (MT-exintr-lst MT).
(defthm flushed-MA-=-MT-final-ISA
    (implies (and (inv MT MA)
                  (not (MT-CMI-p MT))
                  (MAETT-p MT)
                  (MA-state-p MA)
                  (flushed-p MA))
             (equal (proj MA)
                    (ISA-stepn (MT-init-ISA MT)
                               (MT-exintr-lst MT)
                               (MT-len MT))))
  :hints (("goal" :use (:instance ISA-extensionality
                                  (s1 (proj MA))
                                  (s2 (ISA-stepn (MT-init-ISA MT)
                                                 (MT-exintr-lst MT)
                                                 (MT-len MT)))))))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MAIN THEOREM
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This is a main result of our verification work.  The
;; microarchitectural design satisfies a commutative diagram which
;; shows the equivalence of MA implementation and ISA spec; in one
;; path of the diagram, we run the MA design for n clock cycles and
;; then project the final MA state to ISA machine; on the other path,
;; we project the initial MA state to an ISA state, then run the
;; machine for (num-insts MA sigs-lst n) cycles with an appropriate
;; input sequence give by (map-inputs MA sigs-lst n).  Num-insts
;; returns the number of instructions in the MAETT, which is actually
;; the number of instructions executed and committed in the n
;; clock-cycle MA execution.  The appropriate input sequence for the
;; ISA state transitions is constructed by function (map-inputs MA
;; sigs-lst n).
(defthm exintr-correctness-criteria
    (implies (and (MA-state-p MA)
                  (MA-input-listp sigs-lst)
```

```
                  (<= n (len sigs-lst))
                  (flushed-p MA)
                  (flushed-p (MA-stepn MA sigs-lst n))
                  (not (ISA-self-modify-p (proj MA)
                                          (map-inputs MA sigs-lst n)
                                          (num-insts MA sigs-lst n))))
             (equal (proj (MA-stepn MA sigs-lst n))
                    (ISA-stepn (proj MA)
                               (map-inputs MA sigs-lst n)
                               (num-insts MA sigs-lst n))))
  :hints (("Goal" :in-theory (e/d (map-inputs num-insts)
                                  (inv-initial-MT
                                   inv-stepn))
                  :use ((:instance inv-initial-MT)
                        (:instance inv-stepn (MT (init-MT MA))))
                  :restrict ((flushed-MA-=-MT-final-ISA
                              ((MA (MA-STEPN MA SIGS-LST N)))))
                  :do-not-induct t)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Proof of correctness-criteria without interrupts.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defthm trace-exintr-lst-step-trace
    (implies (and (MA-state-p MA)
                  (MA-input-p sigs)
                  (MAETT-p MT)
                  (not (b1p (MA-input-exintr sigs)))
                  (not (b1p (exintr-flag? MA)))
                  (equal (trace-exintr-lst trace)
                         (zero-intr-lst (len trace))))
     (equal (trace-exintr-lst (step-trace trace MT MA sigs ISA spc smc))
            (zero-intr-lst (len (step-trace trace MT MA sigs ISA spc smc))))))

(defthm MT-exintr-lst-MT-step
    (implies (and (MA-state-p MA) (MA-input-p sigs)
                  (MAETT-p MT)
                  (not (b1p (MA-input-exintr sigs)))
                  (not (b1p (exintr-flag? MA)))
                  (equal (MT-exintr-lst MT)
                         (zero-intr-lst (MT-len MT))))
             (equal (MT-exintr-lst (MT-step MT MA sigs))
                    (zero-intr-lst
                     (MT-len (MT-step MT MA sigs)))))
  :hints (("Goal" :in-theory (enable MT-step MT-exintr-lst
                                     MT-len))))

(defthm MT-exintr-lst-MT-stepn-induction
    (implies (and (MA-state-p MA) (MA-input-listp sigs-lst)
                  (MAETT-p MT)
                  (not (b1p (exintr-flag? MA)))
                  (exintr-free-p sigs-lst)
                  (equal (MT-exintr-lst MT)
                         (zero-intr-lst (MT-len MT))))
             (equal (MT-exintr-lst (MT-stepn MT MA sigs-lst n))
                    (zero-intr-lst
                     (MT-len (MT-stepn MT MA sigs-lst n)))))
  :hints (("Goal" :induct (MT-stepn MT MA sigs-lst n)
                  :in-theory (enable lift-b-ops))))

;; The following lemma shows that the ISA input sequence constructed
;; from function map-inputs is a sequence of sigs-lst with 0's if the
;; MA input sequence contains no external interrupts.
```

```
(defthm map-MA-input-lst-=-zero-intr-lst
    (implies (and (MA-state-p MA)
                  (MA-input-listp sigs-lst)
                  (flushed-p MA)
                  (flushed-p (MA-stepn MA sigs-lst n))
                  (exintr-free-p sigs-lst))
             (equal (map-inputs MA sigs-lst n)
                    (zero-intr-lst (num-insts MA sigs-lst n))))
  :hints (("Goal" :in-theory (enable map-inputs num-insts
                                     MA-flushed? lift-b-ops)
                  :do-not-induct t)))

;; The next correctness theorem is an instantiation of the theorem
;; exintr-correctness-criteria.  The lemma shows the equivalence
;; between MA and ISA state transitions, provided that the MA state
;; transitions are not externally interrupted.  The corresponding ISA
;; execution are not externally interrupted, either.
(defthm correctness-criteria
    (implies (and (MA-state-p MA)
                  (MA-input-listp sigs-lst)
                  (<= n (len sigs-lst))
                  (flushed-p MA)
                  (flushed-p (MA-stepn MA sigs-lst n))
                  (exintr-free-p sigs-lst)
                  (not (ISA-self-modify-p (proj MA)
                                          (zero-intr-lst
                                           (num-insts MA sigs-lst n))
                                          (num-insts MA sigs-lst n))))
             (equal (proj (MA-stepn MA sigs-lst n))
                    (ISA-stepn (proj MA)
                               (zero-intr-lst
                                (num-insts MA sigs-lst n))
                               (num-insts MA sigs-lst n))))
  :hints (("Goal" :do-not-induct t)))
```

Bibliography

[AA93] D. Alpert and D. Avon. Architecture of the Pentium microprocessor. *IEEE Micro*, 13(3):11–21, 1993.

[AAC98] The Alpha Architecture Committee. *Alpha Architecture Reference Manual.* Digital Press, Boston, third edition, 1998. `http://www.bh.com/digitalpress`.

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[AL95] M. Agaard and M. Lesser. Reasoning about pipelines with structural hazards. In *Theorem Provers in Circuit Design : theory, practice, and experience*, volume 901 of *LNCS*, pages 13–32. Springer Verlag, 1995.

[BBCZ98] Sergey Berezin, Armin Biere, Edmund Clarke, and Yunshan Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In *Formal Methods in Computer-Aided Design (FMCAD '98)*, volume 1522 of *LNCS*, pages 369–386. Springer Verlag, 1998.

[BD94] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In *Computer-Aided Verification (CAV '94)*, volume 818 of *LNCS*, pages 68–80. Springer Verlag, 1994.

[BH97] Bishop Brock and Warren A. Hunt, Jr. Formally specifying and mechanically verifying programs for the Motorola complex arithmetic processor DSP. In *1997 IEEE International Conference on Computer Design*, pages 31–36. IEEE Computer Society, October 1997.

[BHK94] Bishop C. Brock, Warren A. Hunt, Jr., and Matt Kaufmann. The FM9001 microprocessor proof. Technical Report 86, Computational Logic, Inc., December 1994.

[BKM96] Bishop Brock, Matt Kaufmann, and J Strother Moore. ACL2 theorems about commercial microprocessors. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *LNCS*, pages 275–293. Springer Verlag, 1996.

[BM88] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, Inc., San Diego, California, 1988.

[Bry86] R. K. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[BT90] Alexandre Bronstein and Carolyn L. Talcott. Formal verification of pipelines based on string-functional semantics. In L. J. M. Claesen, editor, *Formal VLSI Correctness Verification, VLSI Design Methods II*, pages 349–366, 1990.

[Bur96] Jerry R. Burch. Techniques for verifying superscalar microprocessors. In *Design Automation Conference (DAC '96)*, pages 552–557, Las Vegas, Nevada, June 1996. ACM Press.

[CAB+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, N.J., 1986.

[CE81] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Workshop on Logics of Programs*, volume 131. Springer Verlag, 1981.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic. *ACM Transactions on Programming Languages and Systems*, 8(2), 1986.

[Coe94] Michael L. Coe. Results from verifying a pipelined microprocessor. Master's thesis, University of Idaho, 1994.

[Coh86] Richard M. Cohen. Proving Gypsy programs. Technical Report 4, Computational Logic, Inc., May 1986.

[Coh87] Avra Cohn. A proof of correctness of the VIPER microprocessor: The first level. Technical Report 104, University of Cambridge, Computer Laboratory, January 1987.

[Cra83] Harvey Cragon. Executable instruction set specification. *Computer Architecture News*, 11(1):25–43, March 1983.

[Cra96] Harvey G. Cragon. *Memory Systems and Pipelined Processors.* Jones and Bartlett Publishers, Sudbury, Massachusetts, 1996.

[Cyr93] David Cyrluk. Microprocessor verification in PVS: A methodology and simple example. Technical Report SRI-CSL-93-12, SRI Computer Science Laboratory, December 1993.

[DP97] W. Damm and A. Pnueli. Verifying out-of-order executions. In D. Probst, editor, *CHARME '97*. Chapman and Hall, 1997.

[GLS90] Jr. Guy L. Steele. *Common Lisp the Language*. Digital Press, second edition, 1990.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, UK, 1993.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *LNCS*. Springer Verlag, 1979.

[HB92] Warren A. Hunt, Jr. and Bishop Brock. A formal HDL and its use in the FM9001 verification. In C. A. R. Hoare and M. J. C. Gordon, editors, *Mechanized Reasoning and Hardware Design*, Prentice-Hall International Series in Computer Science, pages 35–48. Prentice-Hall, Englewood Cliffs, N.J., 1992.

[HGS99] Ravi Hosabettu, Ganesh Gopalakrishnan, and Mandayam Srivas. A proof of correctness of a processor implementing Tomasulo's algorithm without a reorder buffer. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods, 10th IFIP WG10.5 Advanced Research Working Conference, (CHARME '99)*, volume 1703 of *LNCS*, pages 8–22. Springer Verlag, 1999.

[HQR98] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. You assume, we guarantee: Methodology and case studies. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification (CAV '98)*, volume 1427 of *LNCS*, pages 440–451. Springer Verlag, 1998.

[HS99] Warren A. Hunt, Jr. and Jun Sawada. The FM9801 microprocessor verification. *IEEE Micro*, 19(3):47–55, May/June 1999.

[HSG98] Ravi Hosabettu, Mandayam Srivas, and Ganesh Gopalakrishnan. Decomposing the proof of correctness of pipelined microprocessors. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification (CAV '97)*, volume 1427 of *LNCS*, pages 122–134. Springer Verlag, 1998.

[Hun94] Warren A. Hunt, Jr. *FM8501: A Verified Microprocessor*, volume 795 of *LNCS*. Springer Verlag, 1994.

[JDB95] Robert B. Jones, David L. Dill, and Jerry R. Burch. Efficient validity checking for processor verification. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 2–6, 1995.

[Joh91] Mike Johnson. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Kau98] Matt Kaufmann. ACL2 support for verification projects. In C. Kirchner and H. Kirchner, editors, *Proceedings 15th Int'l Conf. Automated Deduction*, volume 1421 of *LNAI*, pages 220–238. Springer Verlag, jul 1998.

[KM96] Matt Kaufmann and J Strother Moore. ACL2: An industrial strength version of nqthm. In *Eleventh Annual Conference on Computer Assurance (COMPASS-96)*, pages 23–34. IEEE Computer Society Press, June 1996.

[KM99] Matt Kaufmann and J Strother Moore. *ACL2: A Computational Logic for Applicative Common Lisp, The User's Manual*. 1999. URL:http://-www.cs.utexas.edu/users/moore/acl2/acl2-doc.html#User's-Manual.

[LL90] Leslie Lamport and Nancy Lynch. Distributed computing models and methods. In *Handbook of Theoretical Computer Science*, volume B, pages 1159–1199. The MIT Press, Cambridge, Ma., 1990.

[McM93] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Press, 1993.

[McM98] K. L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification (CAV '98)*, volume 1427 of *LNCS*, pages 110–121. Springer Verlag, 1998.

[Min97] Mindshare, Inc., Tom Shanley. *Pentium Pro Processor System Architecture.* Addison Wesley Developers Press, 1997. `http://www.aw.com/devpress/`.

[MLK98] J S. Moore, T. Lynch, and M. Kaufmann. A Mechanically Checked Proof of the AMD5$_K$86 Floating-Point Division Program. *IEEE Trans. Comp.*, 47(9):913–926, September 1998. See also URL `http://-devil.ece.utexas.edu/~lynch/divide/divide.html`.

[Moo96] J Strother Moore. *Piton A Mechanically Verified Assembly-Level Language.* Kluwer Academic Publishers, Dordrecht, 1996.

[MSSW94] Cathy May, Ed Silha, Rick Simpson, and Hank Warren, editors. *The PowerPC^{TM} Architecture: A Specification for a New Family of RISC Processors.* Morgan Kaufmann Publishers, Inc., San Francisco, California, second edition, 1994.

[MW97] W. McCune and L. Wos. Otter: The CADE-13 Competition incarnations. *J. Automated Reasoning*, 18(2):211–220, 1997.

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[PH96] David A. Patterson and John L. Hennessey. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, California, second edition, 1996.

[QS82] J.P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proc. of the 5th International Symposium on Programming*, volume 137 of *LNCS*, 1982.

[Rus97] D. Russinoff. A Mechanically Checked Proof of Correctness of the AMD5$_K$86 Floating-Point Square Root Microcode. *Formal Methods in System Design Special Issue on Arithmetic Circuits*, 1997.

[Rus98] D. Russinoff. A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD-K7 Floating-Point Multiplication, Division, and Square Root Instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1:148–200, December 1998.

[Saw99] Jun Sawada. Verification scripts for FM9801 pipelined microprocessor design, 1999. URL:http://www.cs.utexas.edu/users/sawada/FM9801/.

[SB90] Mandayam Srivas and Mark Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, pages 52–64, September 1990.

[SH97] Jun Sawada and Warren A. Hunt, Jr. Trace table based approach for pipelined microprocessor verification. In *Computer Aided Verification (CAV '97)*, volume 1254 of *LNCS*, pages 364–375. Springer Verlag, 1997.

[SH98] Jun Sawada and Warren A. Hunt, Jr. Processor verification with precise exceptions and speculative execution. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification (CAV '98)*, volume 1427 of *LNCS*, pages 135–146. Springer Verlag, 1998.

[SH99] Jun Sawada and Warren A. Hunt, Jr. Results of the verification of a complex pipelined machine model. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods, 10th IFIP WG10.5 Advanced Research Working Conference, (CHARME '99)*, volume 1703 of *LNCS*, pages 313–316. Springer Verlag, 1999.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[SJD98] Jens U. Skakkebæk, Robert B. Jones, and David L. Dill. Formal verification of out-of-order execution using incremental flushing. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification (CAV '98)*, volume 1427 of *LNCS*, pages 98–109. Springer Verlag, 1998.

[SM95] Mandayam K. Srivas and Steven P. Miller. Formal verification of a commercial microprocessor. Technical Report SRI-CSL-95-04, SRI Computer Science Laboratory, July 1995.

[SP85] James E. Smith and Andrew R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *12th Annual International Symposium on Computer Architecture*, pages 36–44, 1985.

[TK94] S. Tahar and R. Kumar. Formal verification of pipeline conflicts in RISC processors. In *European Design Automation Conference (EURO-*

DAC94), pages 285–289, Grenoble, France, September 1994. IEEE Computer Society Press.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

[VB98] Miroslav N. Velev and Randal E. Bryant. Bit-level abstraction in the verification of pipelined microprocessors by correspondence checking. In *Formal Methods in Computer-Aided Design (FMCAD '98)*, volume 1522 of *LNCS*, pages 18–35. Springer Verlag, 1998.

[VB99] Miroslav N. Velev and Randal E. Bryant. Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions to propositional logic. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods, 10th IFIP WG10.5 Advanced Research Working Conference, (CHARME '99)*, volume 1703 of *LNCS*, pages 37–53. Springer Verlag, 1999.

[WB96] Phillip J. Windley and Jerry R. Burch. Mechanically checking a lemma used in an automatic verification tool. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *LNCS*, pages 362–376. Springer Verlag, 1996.

[WC95] Phillip J. Windley and Michael L. Coe. A correctness model for pipelined microprocessors. In *Theorem Provers in Circuit Design : theory, practice and experience*, volume 901 of *LNCS*. Springer Verlag, 1995.

[WGH98] M. M. Wilding, D. A. Greve, and D. S. Hardin. Efficient simulation of formal processor models. Technical report, Advanced Technology

Center, Rockwell Collins Avionics and Communications, Cedar Rapids, IA 52498, 1998. `http://pobox.com/users/hokie/docs/efm.ps`.

[Yu90] Yuan Yu. *Automated Proofs of Object Code for a Widely Used Microprocessor.* PhD thesis, University of Texas at Austin, December 1990.

Vita

Jun Sawada was born in Kyoto, Japan, on the 5th of March 1968, the second son of Mitsu Sawada and Susumu Sawada. He entered Kyoto University, Japan, in 1986, completing a B.S in Mathematics in 1990 and an M.S. in Mathematical Science in 1992. He entered the Graduate School of the University of Texas in 1993, where he was employed as a an assistant instructor and a research assistant. He is a running enthusiast and has completed two marathons during his stay in Austin.

Permanent Address: 6805 Wood Hollow Dr. #349, Austin TX 78731

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin.