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

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 discussed 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)`	$0 \leq \mathtt{v} < 2^{\mathtt{n}}$
`(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 automatically 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` $< \textit{word-name} >$

$\quad$ ($< \textit{field}_1 > \; < \textit{width}_1 > \; < \textit{pos}_1 >$)

$$\vdots$$

$\quad$ ($< \textit{field}_i > \; < \textit{width}_i > \; < \textit{pos}_i >$)

$$\vdots$$

$\quad$ ($< \textit{field}_n > \; < \textit{width}_n > \; < \textit{pos}_n >$))

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

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

$\textbf{if } x \textbf{ then } y \textbf{ else } z \textbf{ fi}$.

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:
$\mathrm{factorial}(x)$
$\stackrel{def}{=}$
$\textbf{if } x \simeq \mathtt{0} \textbf{ then } \mathtt{1}$
$\textbf{ else } x \times \mathrm{factorial}(x - \mathtt{1})$
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)`	$x \in \mathbf{Int}$
`(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))`	$x \not\geq y$
`(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.

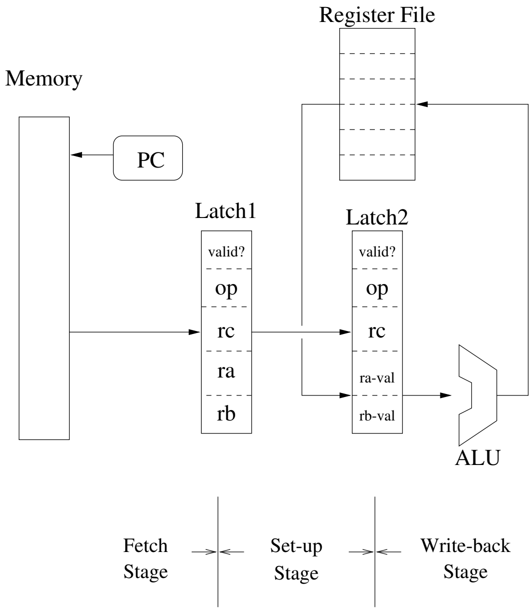

Figure 4.1: The three-stage pipelined machine.

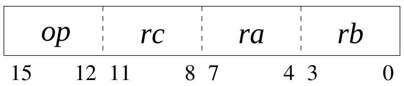

Figure 4.2: The instruction format for the three-stage pipelined machine.

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)
$\stackrel{def}{=}$
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($MA, sig\text{-}list, n$) 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

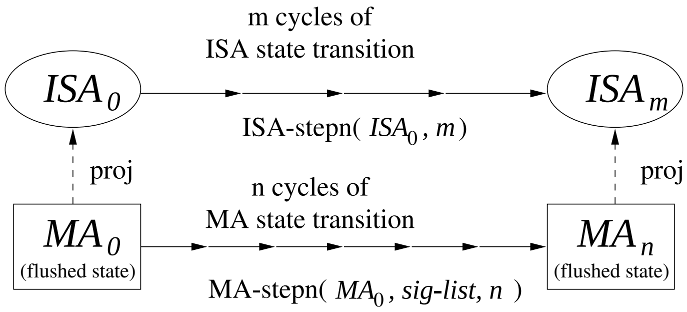

Figure 4.3: The commutative diagram for the pipelined machine.

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*

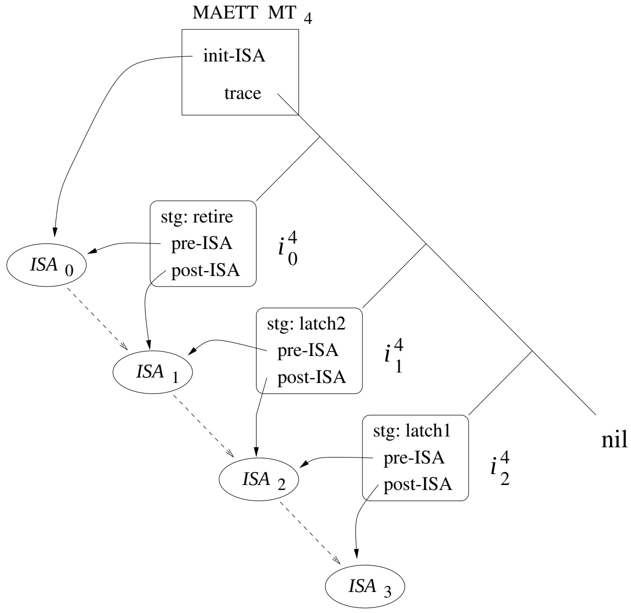

Figure 4.4: The structure of the MAETT intermediate abstraction.

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

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

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

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

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

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

DEFINITION:
INST-rb-val $(i) \stackrel{def}{=}$ read-reg (INST-rb (i), (i.pre-ISA).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:
INST-result (i)
$\stackrel{def}{=}$
ALU-output (INST-op (i), INST-ra-val (i), 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 $(i,\ MA)$
$\stackrel{def}{=}$

$$
\begin{aligned}
& (((MA.\text{latch1}).\text{valid?}) = \mathtt{1}) \\
\wedge\ & (((MA.\text{latch1}).\text{op}) = \text{INST-op}\,(i)) \\
\wedge\ & (((MA.\text{latch1}).\text{rc}) = \text{INST-rc}\,(i)) \\
\wedge\ & (((MA.\text{latch1}).\text{ra}) = \text{INST-ra}\,(i)) \\
\wedge\ & (((MA.\text{latch1}).\text{rb}) = \text{INST-rb}\,(i))
\end{aligned}
$$

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 $(i.\text{stg}) =$ `'latch1` **then** INST-latch1-inv $(i,\ MA)$
 elseif $(i.\text{stg}) =$ `'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 $(MT.\text{trace},\ MA)$

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 $(MT.\text{trace},\ MT.\text{Init-ISA})$

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

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 $(MT,\ MA)$
$\stackrel{def}{=}$

$$
\begin{aligned}
& \text{pc-match-p}\,(MT,\ MA) \\
\wedge\ & \text{regs-match-p}\,(MT,\ MA) \\
\wedge\ & \text{mem-match-p}\,(MT,\ MA) \\
\wedge\ & \text{ISA-chain-p}\,(MT) \\
\wedge\ & \text{MT-INST-invariant}\,(MT,\ MA) \\
\wedge\ & \text{MT-contains-all-insts}\,(MT,\ MA) \\
\wedge\ & \text{MT-in-order-p}\,(MT)
\end{aligned}
$$

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
$(\text{MA-state-p}\,(MA) \wedge (\text{flushed?}\,(MA) = 1)) \rightarrow \text{invariant}\,(\text{init-MT}\,(MA),\ MA)$

Theorem: invariant-step
$(\text{invariant}\,(MT,\ MA) \wedge \text{MAETT-p}\,(MT) \wedge \text{MA-state-p}\,(MA) \wedge \text{MA-sig-p}\,(sig))$
$\rightarrow \text{invariant}\,(\text{MT-step}\,(MT,\ MA,\ sig),\ \text{MA-step}\,(MA,\ sig))$

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

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

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:

$\text{num-insts}\,(MA,\ \textit{sig-list},\ n)$

$\stackrel{def}{=}$

$\text{MT-num-insts}\,(\text{MT-stepn}\,(\text{init-MT}\,(MA),\ MA,\ \textit{sig-list},\ n))$

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 ($i_0^n \ \ldots \ i_{m-1}^n$) 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_k^n.stg = `'retire` for all k such that $0 \leq k < m$. Hence, MT-regs(MT_n) = (i_{m-1}^n.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_{m-1}^n. Since the post-ISA state of i_{m-1}^n 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
$\text{addr-p}\,(x) \leftrightarrow ((x \in \textbf{Int}) \wedge (0 \leq x) \wedge (x < \text{expt}\,(2,\ 16)))$

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

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 $(r1) \land$ rname-p $(r2) \land$ 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 ($r1$) $\wedge$ srname-p ($r2$) $\wedge$ SRF-p (SRF))
$\rightarrow$ (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 ($ad1$) $\wedge$ addr-p ($ad2$) $\wedge$ mem-p (mem))
$\rightarrow$ (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.

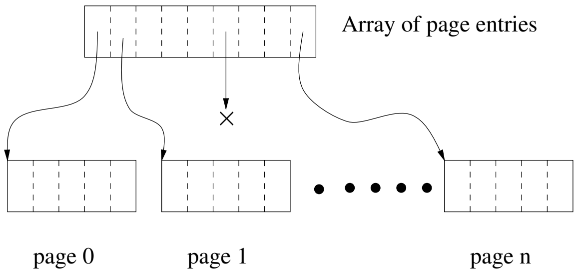

Figure 5.1: Data Structure for Memory Model

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.

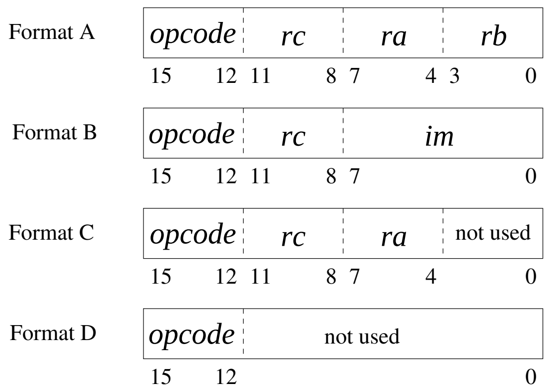

Figure 5.2: Instruction format A, B, C, and D for the FM9801

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*

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

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 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*

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

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*

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

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.

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

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*)
$\stackrel{def}{=}$
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*)
$\stackrel{def}{=}$
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 $\mathtt{10}_{16}$. 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*)
$\stackrel{def}{=}$
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($ISA, intr\text{-}list, n$) defines the ISA state after an n-step execution with a list of external interrupt signals *intr-list*.

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

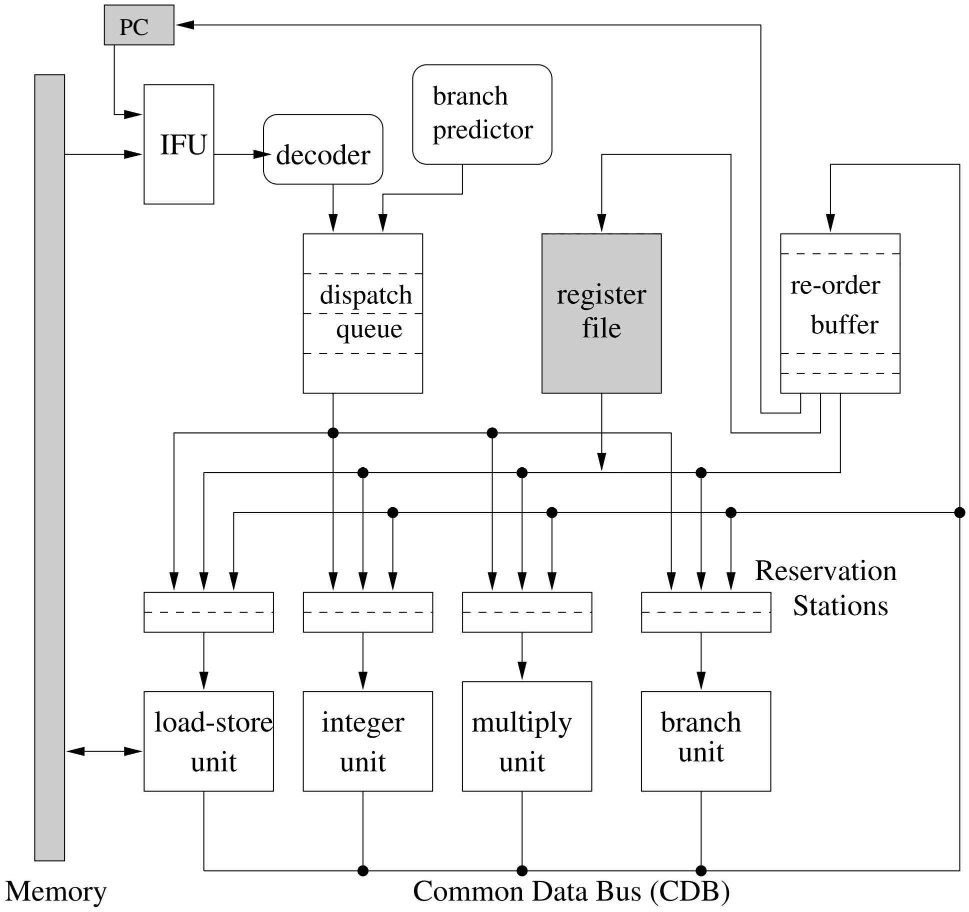

Figure 5.3: Block Diagram of the FM9801.

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

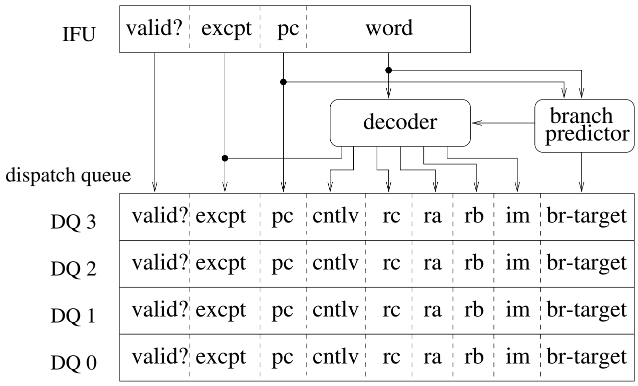

Figure 5.4: The IFU and dispatch queue of the FM9801.

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

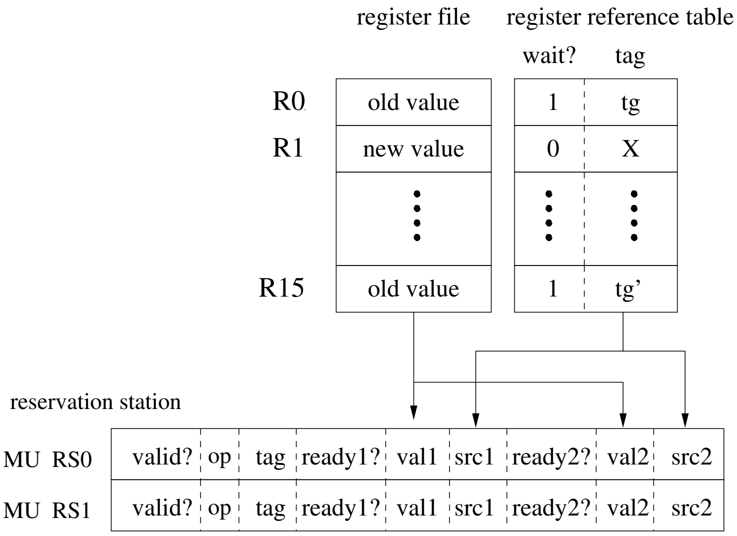

Figure 5.5: The register file, the register reference table and a reservation station.

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,

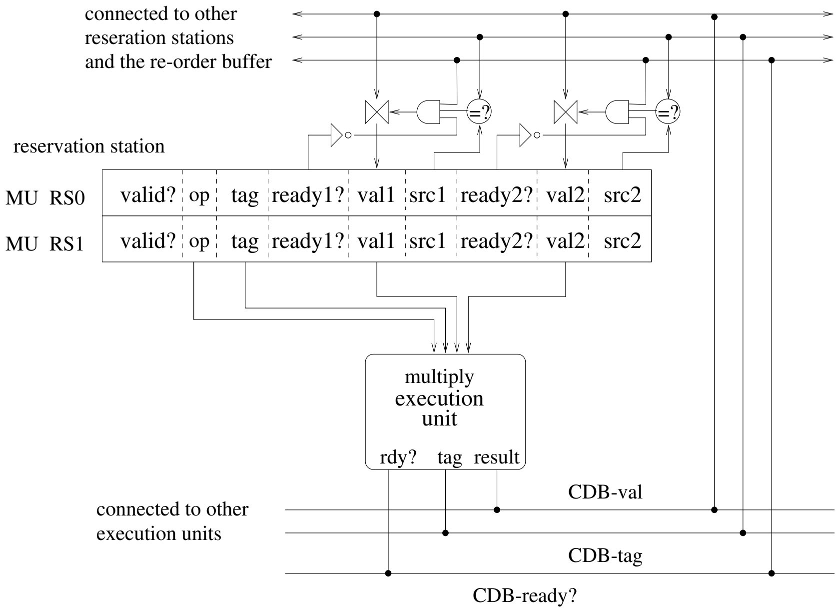

Figure 5.6: A reservation station, the multiply unit and the CDB.

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.

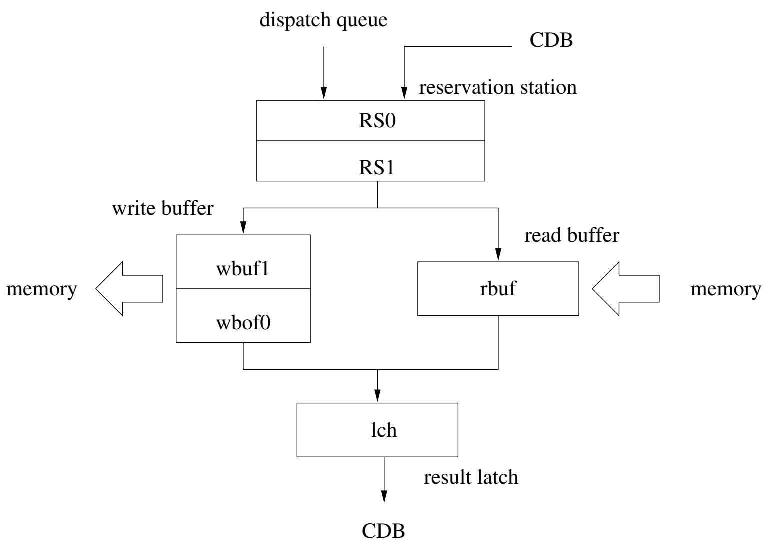

Figure 5.7: The load-store unit of the FM9801.

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, $sigs$)
$\stackrel{def}{=}$
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*)
$\stackrel{def}{=}$
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)
$\stackrel{def}{=}$
if $n \simeq 0$ **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 specification 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.

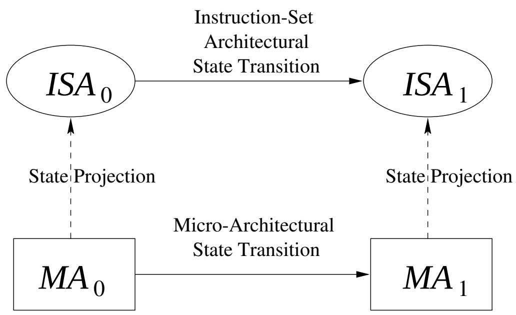

Figure 6.1: Commutative Diagram without Time Abstraction

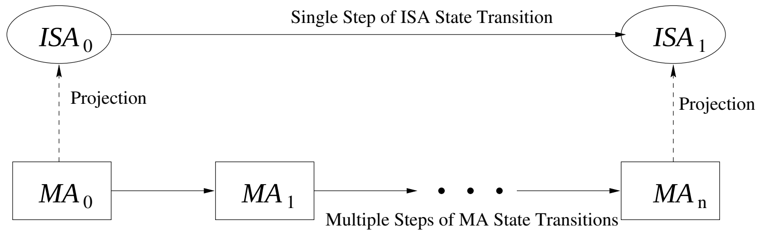

Figure 6.2: Correctness diagram for a machine that takes n cycles to execute a single instruction.

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

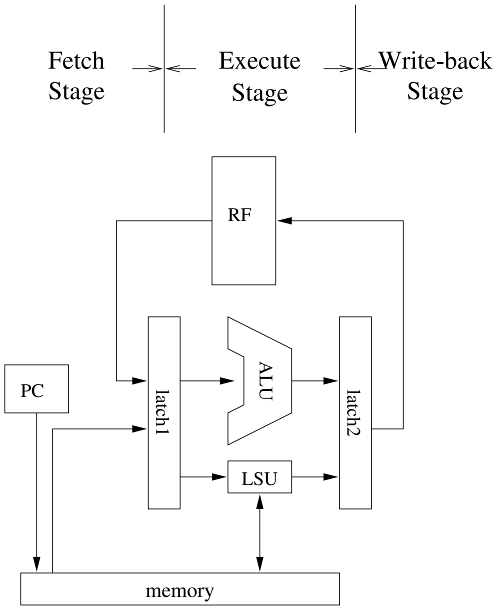

Figure 6.3: An example pipelined machine .

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

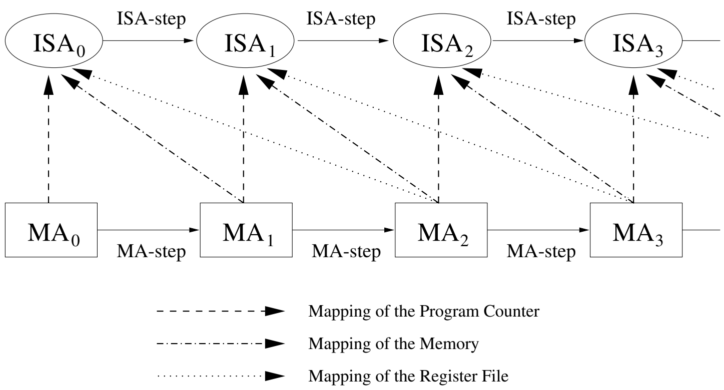

Figure 6.4: Relation of ISA and MA states with timing delay.

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 function 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,

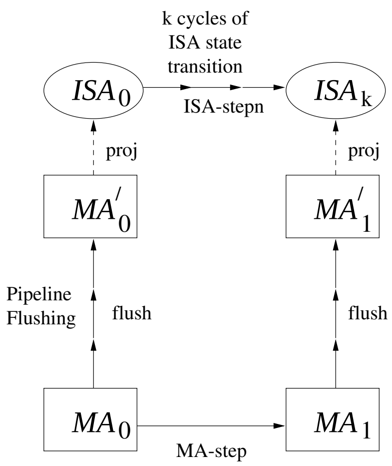

Figure 6.5: Burch and Dill's Flushing Diagram.

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

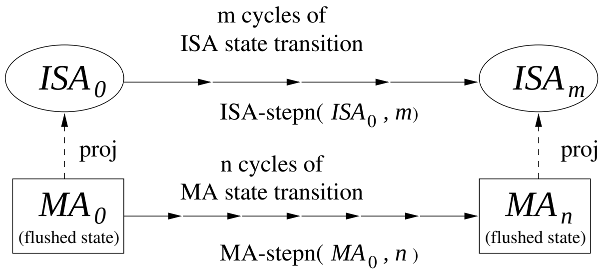

Figure 6.6: Correctness diagram. This diagram compares n cycles of MA execution and m cycles of ISA execution, where m is the number of instructions executed during the MA execution.

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*

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

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

```
STORE R0, <addr>
IO_REQUEST
```

(a)

```
STORE R0, <addr>
NOP
NOP
IO_REQUEST
```

(b)

```
STORE R0, <addr>
SYNC
IO_REQUEST
```

(c)

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

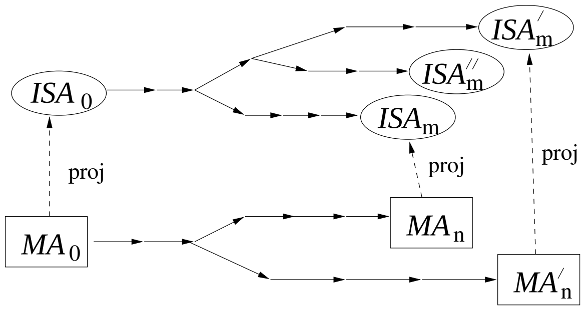

Figure 6.8: Correspondence between the ISA and the MA with exceptions.

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 $\text{proj}(MA_0) = ISA_0$ and $\text{proj}(MA_n) = ISA_m$. 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($ISA_0, \textit{intr-list}, n$), which checks whether the ISA execution corresponding to ISA-stepn($ISA_0, \textit{intr-list}, n$) 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*

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

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 $MA' = \text{MA-step}(MA, \textit{sigs})$ 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 $\text{MA-stepn}(MA_0, \textit{sig-list}, n)$ 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 $\text{MT-step}(MT, MA, \textit{sigs})$ 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 $MA' = \text{MA-step}(MA, \textit{sigs})$. Let MT be the MAETT of MA. Then, $\text{MT-step}(MT, MA, \textit{sigs})$ 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 $\text{MT-stepn}(MT_0, MA_0, \textit{sig-list}, n)$ defines the MAETT for the reachable state $\text{MA-stepn}(MA_0, \textit{sig-list}, n)$.

DEFINITION:
MT-stepn (MT, MA, *sig-list*, n)
$\stackrel{def}{=}$
if $n \simeq 0$ **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, $MT.\mathrm{trace} = (i_0 \ \ i_1 \ \ i_2)$.

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$(MA_0, \textit{sig-list}, n)$ and its MAETT is defined as MT-stepn(MT-init$(MA_0), MA_0, \textit{sig-list}, n)$. 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**.

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

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, $i'_k =$ 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)` → `'(DQ1)` → `'(DQ0)` → `'(IU RS0)` → `'(complete)` → `'(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.

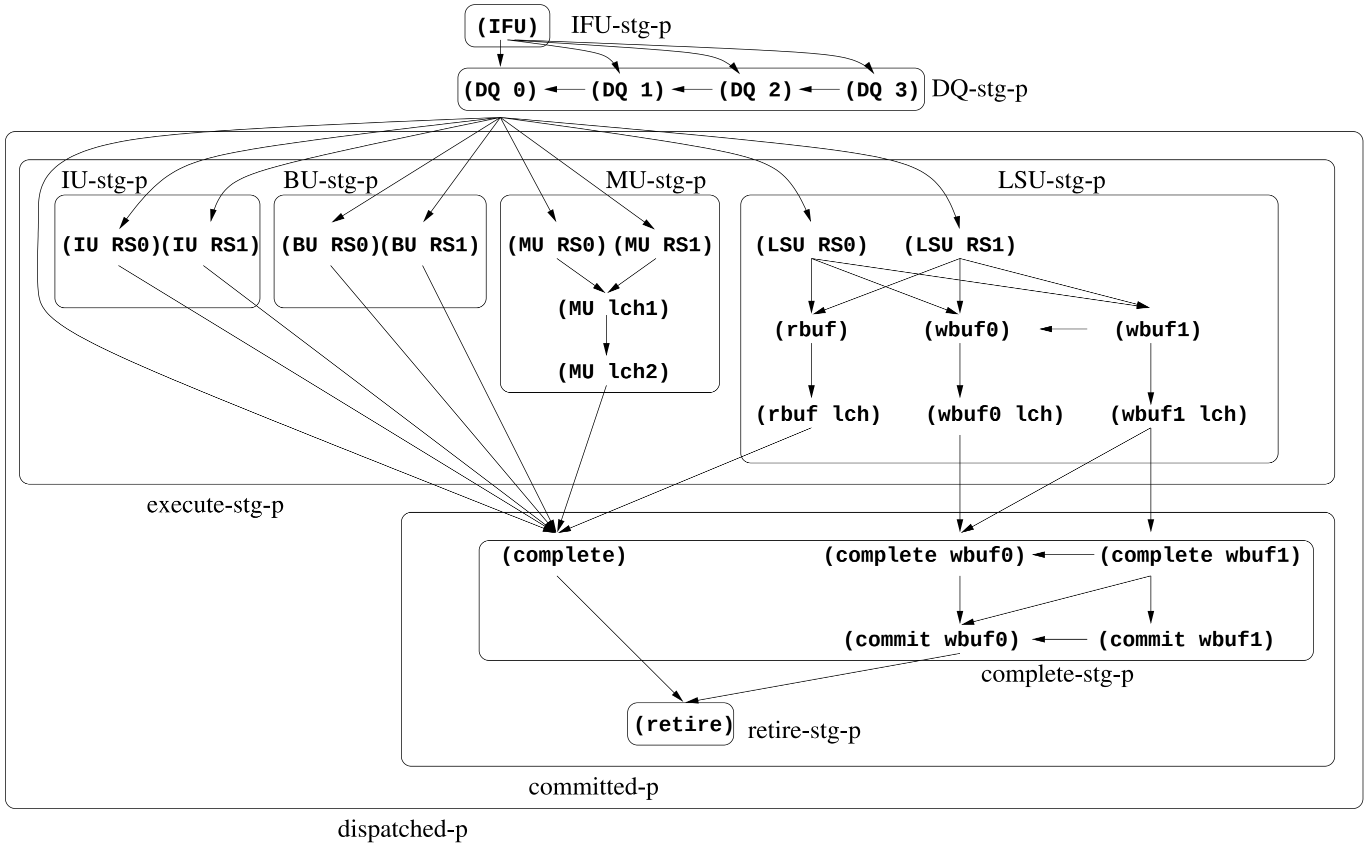

Figure 7.3: Stages of Instructions in the FM9801. Boxes surround the stages satisfying the labeling predicates.

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$(ISA_0, i.\text{exintr?})$, 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 *specultv?* 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 $\neg$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)
$\stackrel{def}{=}$
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)
$\stackrel{def}{=}$
$\quad$(INST-store? $(i) = 1$)
$\wedge$ (INST-store-addr $(i) = ((j.\text{pre-ISA}).\text{pc})$)

$\wedge$ (INST-excpt?(i) $\neq$ 1)
$\wedge$ ((i.exintr?) $\neq$ 1)
$\wedge$ ((j.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.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* ∈ *list*, returns the tail of *list* beginning with the first occurrence of *elm* if *elm* is a member of *list*.

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

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

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
$(\quad \mathrm{inv}\,(MT,\ MA)$
$\wedge\ \text{MAETT-p}\,(MT)$
$\wedge\ \text{MA-state-p}\,(MA)$
$\wedge\ (i$ **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 $\mathrm{inv}(MT, MA)$ as hypothesis. This predicate is our invariant condition that will be defined in the next chapter. In this chapter, consider $\mathrm{inv}(MT, MA)$ as the well-formedness predicate for the MAETT MT.

THEOREM: INST-in-order-p-total

$((i \in_{\mathrm{MT}} MT) \wedge (j \in_{\mathrm{MT}} MT) \wedge (\neg\ (j \textbf{ precedes } i \textbf{ in } MT)) \wedge (i \neq j))$

$\rightarrow (i \textbf{ precedes } j \textbf{ in } MT)$

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 $(s,\ MT.\mathrm{trace})$

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

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$(i.\mathrm{stg}, MT)$ 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 $(i.\mathrm{stg},\ MT))$
$\rightarrow$ (INST-at-stg $(i.\mathrm{stg},\ MT) = i)$

We define similar functions and predicates that take a list of stages as an argument. The function INST-at-stgs$(s\text{-}list, MT)$ 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$(s\text{-}list, MT)$ is true iff exactly one instruction is at one of the stages in *s-list*, and no-INST-at-stgs$(s\text{-}list, MT)$ 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 $(i,\ r,\ MT.\mathrm{trace})$

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 $(i,\ r,\ MT.\mathrm{trace})$

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 $(i,\ r,\ MT) \to$ 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 $(i,\ r,\ MT))$
$\to$ (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 $(i,\ r,\ MT) \neq j))$
$\to$ (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 $(i) \wedge (i \in_{\mathrm{MT}} MT) \wedge ((i.\text{specultv?}) \neq 1) \wedge ((i.\text{modified?}) \neq 1))$
$\wedge$ exist-LRM-before-p $(i,\ r,\ MT)$
$\wedge$ $(\neg$ committed-p (LRM-before $(i,\ r,\ MT))))$
$\to$ (INST-dest-val (LRM-before $(i,\ r,\ MT)) =$ read-reg $(r,\ (i.\text{pre-ISA}).\text{RF}))$

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)*

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

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,*

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

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.*

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

The predicate $\mathrm{inv}(MT, MA)$ satisfies the following two lemmas.

Theorem 2

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

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,*

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

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 $\mathrm{inv}(MT, MA)$ is an *invariant under the constraint* $\neg$MT-CMI-p(MT) [LL90]; that is, $\mathrm{inv}(MT, MA)$ 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 $\neg$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)
∧ MA-state-p (MA)
∧ MA-input-p ($sigs$)
∧ weak-INV (MT))
→ 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 (¬ dispatched-p (car (*trace*))) ∧ no-dispatched-INST-p (cdr (*trace*))
fi

DEFINITION:
no-commit-INST-p (*trace*)
$\stackrel{def}{=}$
if endp (*trace*) **then t**
else (¬ committed-p (car (*trace*))) ∧ 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 ¬ dispatched-p (car (*trace*)) **then** no-dispatched-INST-p (cdr (*trace*))
elseif ¬ committed-p (car (*trace*)) **then** no-commit-INST-p (cdr (*trace*))
else t
fi
∧ 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) ∧ MAETT-p (MT) ∧ MA-state-p (MA))
∧ (($i \in_{\mathrm{MT}} MT$) ∧ ($j \in_{\mathrm{MT}} MT$))
∧ dispatched-p (i)
∧ (¬ dispatched-p (j)))
→ (i **precedes** j **in** MT)

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

$$\begin{array}{l}
\quad\;\; \wedge\;\; \text{committed-p}\,(i) \\
\quad\;\; \wedge\;\; (\neg\ \text{committed-p}\,(j))) \\
\rightarrow (i\ \textbf{precedes}\ j\ \textbf{in}\ MT)
\end{array}$$

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

$$\begin{array}{l}
\quad (\quad \text{inv}\,(MT,\ MA) \\
\quad\;\; \wedge\;\; \text{MAETT-p}\,(MT) \\
\quad\;\; \wedge\;\; \text{MA-state-p}\,(MA) \\
\quad\;\; \wedge\;\; \text{DQ-stg-p}\,(i.\text{stg}) \\
\quad\;\; \wedge\;\; \text{DQ-stg-p}\,(j.\text{stg}) \\
\quad\;\; \wedge\;\; (i\ \textbf{precedes}\ j\ \textbf{in}\ MT)) \\
\rightarrow (\text{DQ-stg-idx}\,(i.\text{stg}) < \text{DQ-stg-idx}\,(j.\text{stg}))
\end{array}$$

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, $\mathit{flg} = 0$ and $\mathit{head} = \mathit{tail}$.

We define a relation $rix1 <_{\text{tag}} rix2$ **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 $ROB\text{-}\mathit{flg}$, $ROB\text{-}\mathit{head}$, and $ROB\text{-}\mathit{tail}$, respectively. Thus, the relation $rix1 <_{\text{tag}} rix2$ **in** MT is defined as follows.

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

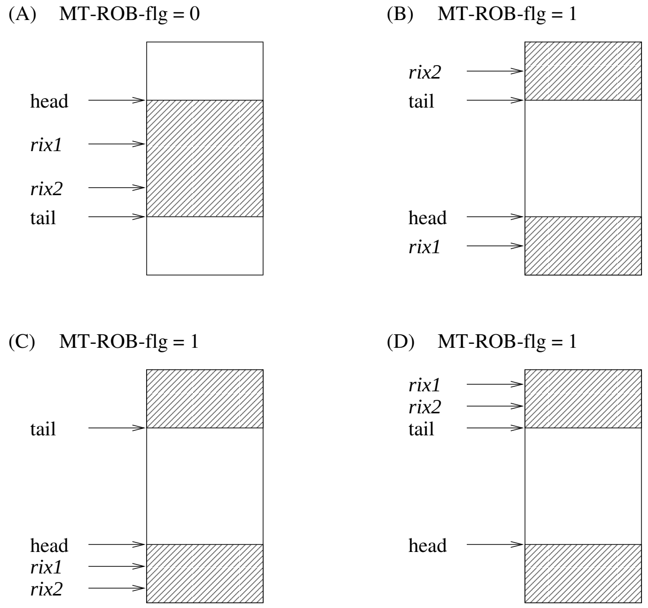

Figure 8.1: Order of Instructions in the Reorder Buffer

$\lor$ $(((MT.\text{ROB-head}) \leq \mathit{rix1}) \land (\mathit{rix1} < \mathit{rix2}))$
$\lor$ $((\mathit{rix1} < \mathit{rix2}) \land (\mathit{rix2} < (MT.\text{ROB-tail})))$
else $\mathit{rix1} < \mathit{rix2}$
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 $rix1 <_{\mathrm{MT}} rix2$ **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 $<_{\mathrm{MT}}$ j.tag **in** MT.

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

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

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:
in-order-LSU-INST-p $(MT,\ MA)$
$\stackrel{def}{=}$
in-order-LSU-RS-p $(MT.\mathrm{trace},\ MA)$

$\wedge$ 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 $((MA.\text{IFU}).\text{valid?}) = \texttt{1}$ **then** uniq-INST-at-stg $(\texttt{'(IFU)},\ MT)$
else no-INST-at-stg $(\texttt{'(IFU)},\ MT)$
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 $(\text{nth-robe}\,(idx,\ MA.\text{ROB}).\text{valid?}) = \texttt{1}$ **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 $idx \simeq \texttt{0}$ **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 $(\texttt{8},\ MT,\ MA)$

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$(idx, MA.\text{ROB})$ returns the state of the idx'th entry of reorder buffer in state MA. If its busy flag, nth-robe$(idx, MA.\text{ROB})$.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$(idx, MA.\text{ROB})$ 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.

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

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.

$$
\begin{array}{l}
\textsc{Theorem}\text{: INST-in-order-p-INST-start-specultv} \\
\quad (\quad \mathrm{inv}\,(MT,\ MA) \\
\quad\ \land\ \text{MAETT-p}\,(MT) \\
\quad\ \land\ \text{MA-state-p}\,(MA) \\
\quad\ \land\ (\text{INST-start-specultv?}\,(i) = 1) \\
\quad\ \land\ (i\ \textbf{precedes}\ j\ \textbf{in}\ MT)) \\
\to ((j.\text{specultv?}) = 1)
\end{array}
$$

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

$$
\begin{array}{l}
\quad (\quad (\text{inv}\,(MT,\, MA) \wedge \text{MAETT-p}\,(MT) \wedge \text{MA-state-p}\,(MA)) \\
\quad\ \wedge\ (\text{INST-p}\,(i) \wedge (i \in_{\text{MT}} MT)) \\
\quad\ \wedge\ \text{committed-p}\,(i)) \\
\rightarrow ((i.\text{specultv?}) = \mathtt{0})
\end{array}
$$

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

$$
\begin{array}{l}
(\quad (\text{inv}\,(MT,\ MA) \land \text{MAETT-p}\,(MT) \land \text{MA-state-p}\,(MA)) \\
\quad \land\ (\text{INST-p}\,(i) \land (i \in_{\text{MT}} MT)) \\
\quad \land\ (\neg\ \text{retire-stg-p}\,(i.\text{stg}))) \\
\to ((i.\text{exintr?}) = \mathtt{0})
\end{array}
$$

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) $\land$ 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 $(i.\text{stg})$ **then** IFU-INST-inv $(i,\ MA)$
elseif DQ-stg-p $(i.\text{stg})$ **then** DQ-INST-inv $(i,\ MA)$
elseif execute-stg-p $(i.\text{stg})$ **then** execute-INST-inv $(i,\ MA)$
elseif complete-stg-p $(i.\text{stg})$ **then** complete-INST-inv $(i,\ MA)$
elseif commit-stg-p $(i.\text{stg})$ **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 $(i,\ MA)$
$\stackrel{def}{=}$

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

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 $(MT,\ MA)$
∧ ((i.stg) = '(MU RS0))
∧ ((((MA.MU).RS0).ready1?) $\neq$ 1)
∧ ((i.specultv?) $\neq$ 1)
∧ ((i.modified?) $\neq$ 1)
∧ MAETT-p (MT)

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

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? $(MT) = 1$) $\lor$ (MT-modified-at-dispatch? $(MT) = 1$)
then t
elseif (reg-tbl-nth $(r, (MA.\mathrm{DQ}).\text{reg-tbl}).\text{wait?}) = 1$
then exist-LRM-in-ROB-p (r, MT)
$\land$ ((LRM-in-ROB $(r, MT).\mathrm{tag}) = (\text{reg-tbl-nth}\,(r, (MA.\mathrm{DQ}).\text{reg-tbl}).\mathrm{tag}))$
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 (`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 $(MT,\ MA)$
$\wedge$ $((i.\text{stg}) =$ `'(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

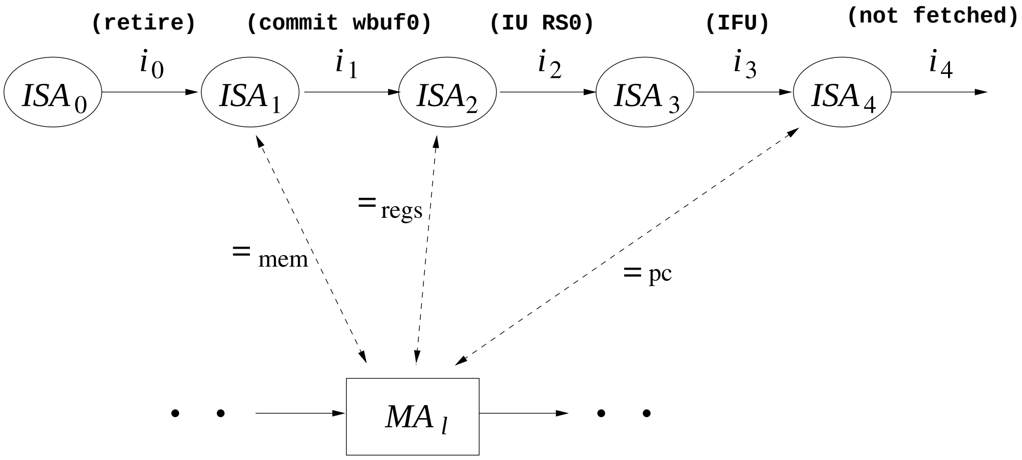

Figure 8.2: Relation between the states of the programmer visible components in the ISA and the MA.

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 $(MT) = (MA.\text{pc})$)

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 $(MT) = (MA.\text{RF})$

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

elseif $\neg$ 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 $\neg$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 $(MA.\mathrm{DQ})$
$\wedge$ consistent-ROB-p $(MA.\mathrm{ROB})$
$\wedge$ consistent-LSU-p $(MA.\mathrm{LSU})$

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 $(MT,\ MA)$
$\stackrel{def}{=}$
 $(((MA.\mathrm{ROB}).\mathrm{flg}) = (MT.\mathrm{ROB\text{-}flg}))$
$\wedge$ $(((MA.\mathrm{ROB}).\mathrm{head}) = (MT.\mathrm{ROB\text{-}head}))$
$\wedge$ $(((MA.\mathrm{ROB}).\mathrm{tail}) = (MT.\mathrm{ROB\text{-}tail}))$
$\wedge$ $((MT.\mathrm{DQ\text{-}len}) \leq 4)$
$\wedge$ 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 $\text{RF-match-p}(MT, MA)$. Additional assumptions are type predicates.

Theorem: RF-match-p-preserved

$$\begin{array}{l} (\quad (\text{MAETT-p}\,(MT) \wedge \text{MA-state-p}\,(MA) \wedge \text{MA-input-p}\,(sigs)) \\ \quad \wedge\ \text{inv}\,(MT,\ MA) \\ \quad \wedge\ (\neg\ \text{MT-CMI-p}\,(\text{MT-step}\,(MT,\ MA,\ sigs)))) \\ \rightarrow \text{RF-match-p}\,(\text{MT-step}\,(MT,\ MA,\ sigs),\ \text{MA-step}\,(MA,\ sigs)) \end{array}$$

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

$$(\text{MA-state-p}\,(MA) \wedge (\text{MA-flushed?}\,(MA) = 1)) \rightarrow \text{inv}\,(\text{init-MT}\,(MA),\ MA)$$

THEOREM: inv-step

$$\begin{array}{l}(\quad (\text{MAETT-p}\,(MT) \wedge \text{MA-state-p}\,(MA) \wedge \text{MA-input-p}\,(sigs)) \\ \quad \wedge\ \text{inv}\,(MT,\ MA) \\ \quad \wedge\ (\neg\ \text{MT-CMI-p}\,(\text{MT-step}\,(MT,\ MA,\ sigs)))) \\ \to \text{inv}\,(\text{MT-step}\,(MT,\ MA,\ sigs),\ \text{MA-step}\,(MA,\ sigs))\end{array}$$

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

$$\begin{array}{l}(\quad (\text{MAETT-p}\,(MT) \wedge \text{MA-state-p}\,(MA) \wedge \text{MA-input-p}\,(sigs)) \\ \quad \wedge\ \text{inv}\,(MT,\ MA) \\ \quad \wedge\ (\neg\ \text{MT-CMI-p}\,(\text{MT-step}\,(MT,\ MA,\ sigs)))) \\ \to \text{MT-INST-inv}\,(\text{MT-step}\,(MT,\ MA,\ sigs),\ \text{MA-step}\,(MA,\ sigs))\end{array}$$

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

$$
\begin{array}{l}
\quad (\quad (\text{MAETT-p}\,(MT) \wedge \text{MA-state-p}\,(MA) \wedge \text{MA-input-p}\,(\mathit{sigs})) \\
\quad\ \wedge\ (\text{INST-p}\,(i) \wedge (i \in_{\text{MT}} MT)) \\
\quad\ \wedge\ \text{inv}\,(MT,\ MA) \\
\quad\ \wedge\ (\neg\ \text{MT-CMI-p}\,(\text{MT-step}\,(MT,\ MA,\ \mathit{sigs}))) \\
\quad\ \wedge\ \text{MT-no-jmp-exintr-before}\,(i,\ MT,\ MA,\ \mathit{sigs}) \\
\quad\ \wedge\ (\text{INST-exintr-now?}\,(i,\ MA,\ \mathit{sigs}) \neq \mathtt{1})) \\
\rightarrow \text{INST-inv}\,(\text{step-INST}\,(i,\ MT,\ MA,\ \mathit{sigs}),\ \text{MA-step}\,(MA,\ \mathit{sigs}))
\end{array}
$$

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

$$
\begin{array}{l}
\quad (\quad \text{inv}\,(MT,\ MA) \\
\quad\ \wedge\ \text{MAETT-p}\,(MT) \\
\quad\ \wedge\ \text{MA-state-p}\,(MA) \\
\quad\ \wedge\ (i \in_{\text{MT}} MT) \\
\quad\ \wedge\ \text{IFU-stg-p}\,(i.\text{stg}) \\
\quad\ \wedge\ ((i.\text{specultv?}) \neq \mathtt{1}) \\
\quad\ \wedge\ ((i.\text{modified?}) \neq \mathtt{1})) \\
\rightarrow (((MA.\text{IFU}).\text{word}) = \text{INST-word}\,(i))
\end{array}
$$

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
$$
\begin{array}{l}
(\quad (\text{inv}\,(MT,\ MA) \land \text{MAETT-p}\,(MT) \land \text{MA-state-p}\,(MA)) \\
\ \land\ \text{INST-writeback-p}\,(j) \\
\ \land\ ((j.\text{specultv?}) \neq \mathtt{1}) \\
\ \land\ ((j.\text{modified?}) \neq \mathtt{1}) \\
\ \land\ (\neg\ \text{INST-excpt-detected-p}\,(j)) \\
\ \land\ (\text{CDB-ready?}\,(MA) = \mathtt{1})) \\
\to (\text{CDB-val}\,(MA) = \text{INST-dest-val}\,(j))
\end{array}
$$

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*
$$
\begin{array}{l}
(\quad (\text{inv}\,(MT,\ MA) \land \text{MAETT-p}\,(MT) \land \text{MA-state-p}\,(MA)) \\
\ \land\ (\text{INST-p}\,(i) \land (i \in_{\text{MT}} MT)) \\
\ \land\ ((i.\text{stg}) = \texttt{'(MU RS0)}) \\
\ \land\ ((i.\text{specultv?}) \neq \mathtt{1}) \\
\ \land\ ((i.\text{modified?}) \neq \mathtt{1}) \\
\ \land\ (\text{CDB-ready?}\,(MA) = \mathtt{1}) \\
\ \land\ (\text{CDB-tag}\,(MA) = (((MA.\text{MU}).\text{RS0}).\text{src1})) \\
\ \land\ ((((MA.\text{MU}).\text{RS0}).\text{ready1?}) \neq \mathtt{1})) \\
\to (\text{CDB-val}\,(MA) = \text{INST-src-val1}\,(i))
\end{array}
$$

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:

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

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

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

□

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.

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

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

$$
\begin{array}{l}
\quad (\quad (\text{inv}\,(MT,\ MA) \land \text{MAETT-p}\,(MT) \land \text{MA-state-p}\,(MA) \land \text{MA-input-p}\,(sigs)) \\
\quad\ \land\ ((i \in_{\text{MT}} MT) \land \text{INST-p}\,(i)) \\
\quad\ \land\ ((i.\text{specultv?}) \neq \texttt{1}) \\
\quad\ \land\ ((i.\text{modified?}) \neq \texttt{1}) \\
\quad\ \land\ ((i.\text{stg}) = \texttt{'(LSU rbuf)}) \\
\quad\ \land\ (\text{release-rbuf?}\,(MA.\text{LSU},\ MA,\ sigs) = \texttt{1}) \\
\quad\ \land\ (\text{LSU-address-match?}\,(MA.\text{LSU}) = \texttt{1})) \\
\to (\text{LSU-forward-wbuf}\,(MA.\text{LSU}) = \text{INST-dest-val}\,(i))
\end{array}
$$

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

$$
\begin{array}{l}
\quad (\quad (\text{inv}\,(MT,\ MA) \land \text{MA-state-p}\,(MA)) \\
\quad\ \land\ (\text{INST-p}\,(i) \land (i \in_{\text{MT}} MT)) \\
\quad\ \land\ ((i.\text{specultv?}) \neq \texttt{1}) \\
\quad\ \land\ ((i.\text{modified?}) \neq \texttt{1}) \\
\quad\ \land\ ((i.\text{stg}) = \texttt{'(LSU rbuf)}) \\
\quad\ \land\ (\text{LSU-address-match?}\,(MA.\text{LSU}) \neq \texttt{1})) \\
\to (\neg\ \text{exist-non-retired-LMM-before-p}\,(i,\ \text{INST-load-addr}\,(i),\ MT))
\end{array}
$$

THEOREM: read-mem-when-no-active-mem-modifier-before

$$
\begin{array}{l}
\quad (\quad (\text{inv}\,(MT,\ MA) \land \text{MAETT-p}\,(MT) \land \text{MA-state-p}\,(MA)) \\
\quad\ \land\ \text{addr-p}\,(ad) \\
\quad\ \land\ (i \in_{\text{MT}} MT) \\
\quad\ \land\ \text{INST-p}\,(i) \\
\quad\ \land\ (\neg\ \text{retire-stg-p}\,(i.\text{stg})) \\
\quad\ \land\ (\neg\ \text{exist-non-retired-LMM-before-p}\,(i,\ ad,\ MT))) \\
\to (\text{read-mem}\,(ad,\ MA.\text{mem}) = \text{read-mem}\,(ad,\ (i.\text{pre-ISA}).\text{mem}))
\end{array}
$$

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.

$$
\begin{array}{l}
\textsc{Theorem:}\ \text{read-mem-INST-load-addr-INST-dest-val} \\
\quad (\quad (\text{inv}\,(MT,\, MA) \wedge \text{MAETT-p}\,(MT) \wedge \text{MA-state-p}\,(MA)) \\
\quad\ \ \wedge\ \ ((i \in_{\text{MT}} MT) \wedge \text{INST-p}\,(i)) \\
\quad\ \ \wedge\ \ ((i.\text{specultv?}) \neq \mathtt{1}) \\
\quad\ \ \wedge\ \ ((i.\text{modified?}) \neq \mathtt{1}) \\
\quad\ \ \wedge\ \ ((i.\text{stg}) = \mathtt{'(LSU\ rbuf)}) \\
\quad\ \ \wedge\ \ (\text{LSU-address-match?}\,(MA.\text{LSU}) \neq \mathtt{1})) \\
\to (\text{read-mem}\,(\text{INST-load-addr}\,(i),\, MA.\text{mem}) = \text{INST-dest-val}\,(i))
\end{array}
$$

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 $\mathrm{inv}(MT_k, MA_k)$ 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, $\mathrm{proj}(MA_0)$, of the initial MA state MA_0. In order to prove our criterion, we need to show that

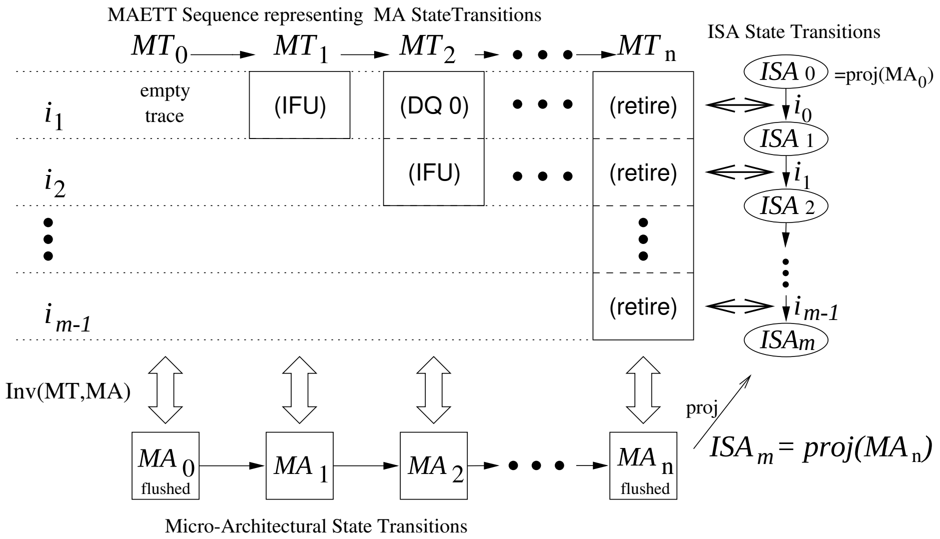

Figure 9.1: Relation between ISA, MA, and MAETT sequences.

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 example, 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,*

$$\text{inv}(MT, MA) \land \text{MT-all-retired-p}(MT) \rightarrow \neg\text{MT-specultv-p}(MT)\ .$$

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,*

$$\text{MT-all-retired-p}(MT) \land \neg\text{MT-CMI-p}(MT) \rightarrow \neg\text{MT-self-modify-p}(MT)\ .$$

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,*

$$\begin{aligned}&\neg\text{ISA-self-modify-p}(MT.\text{init-ISA}, \text{MT-exintr-lst}(MT), \text{MT-len}(MT))\\ &\rightarrow \neg\text{MT-CMI-p}(MT)\ .\end{aligned}$$

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

$$\begin{array}{rl} (& \text{inv}\,(MT,\, MA) \\ \wedge & (\neg\ \text{MT-specultv-p}\,(MT)) \\ \wedge & (\neg\ \text{MT-self-modify-p}\,(MT))) \\ \rightarrow & (\text{MT-pc}\,(MT) = MT.\text{pc}) \end{array}$$

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*

$$\begin{array}{rcl} m & = & \text{MT-len}(MT_n) \\ ISA_m & = & \text{ISA-stepn}(MT_n.\text{init-ISA}, \text{MT-exintr-lst}(MT_n), m) \end{array}$$

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*

$$\begin{aligned} m &= \text{MT-len}(MT_n) \\ \mathit{ISA}_m &= \text{ISA-stepn}(MT_n.\text{init-ISA}, \text{MT-exintr-lst}(MT_n), m) \end{aligned}$$

Then,

$$\text{MT-all-retired-p}(MT_n) \rightarrow \text{MT-RF}(MT_n) = \mathit{ISA}_m.\text{RF} \ .$$

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*

$$\begin{aligned} m &= \text{MT-len}(MT_n) \\ \mathit{ISA}_m &= \text{ISA-stepn}(MT_n.\text{init-ISA}, \text{MT-exintr-lst}(MT_n), m) \end{aligned}$$

Then,

$$\text{MT-all-retired}(MT_n) \rightarrow \text{MT-SRF}(MT_n) = \mathit{ISA}_m.\text{SRF} \ .$$

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$(MA_0), \textit{intr-list}, m)$, 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 $\neg$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 $\neg$MT-specultv-p(MT_n) from Lemma 4, and we can prove $\neg$MT-self-modify-p(MT_n) from Lemma 6.

Using Lemmas 8 and 9, we have

$$MA_n.\mathrm{pc} = \text{MT-pc}(MT_n) = \mathit{ISA}_m.\mathrm{pc}\ .$$

Since MT-all-retired-p(MT_n) holds, we obtain MT-RF(MT_n) $=$ ISA_m.RF from Lemma 11. With Lemma 10,

$$MA_n.\mathrm{RF} = \text{MT-RF}(MT_n) = \mathit{ISA}_m.\mathrm{RF}\ .$$

Similarly, from Lemmas 12 and 13, we obtain

$$MA_n.\mathrm{SRF} = \text{MT-SRF}(MT_n) = \mathit{ISA}_m.\mathrm{SRF},$$

and from Lemmas 14 and 15,

$$MA_n.\mathrm{mem} = \text{MT-mem}(MT_n) = \mathit{ISA}_m.\mathrm{mem}\ .$$

From the four equalities above and the definition of proj(MA), we have

$$\mathrm{proj}(MA_n) = \mathit{ISA}_m\ .$$

From the definition of ISA_m,

$$\mathrm{proj}(MA_n) = \text{ISA-stepn}(\mathrm{proj}(MA_0), \mathit{sig\text{-}list}, m) \quad \square$$

The witness functions for our correctness criterion are:

$$W_N(MA, \mathit{sig\text{-}list}, n) = \text{MT-len}(\text{MT-stepn}(\text{MT-init}(MA), MA, \mathit{sig\text{-}list}, n))$$
$$W_{sig}(MA, \mathit{sig\text{-}list}, n) = \text{MT-exintr-lst}(\text{MT-stepn}(\text{MT-init}(MA), MA, \mathit{sig\text{-}list}, n))\ .$$

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 verification. 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 potential 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 $0 \leq i < n$,*

$$\text{proj}(\text{flush}(\text{MA-step}(MA_i))) = \text{ISA-stepn}(\text{proj}(\text{flush}(MA_i)), k_i) \tag{A.1}$$

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:

$$\begin{array}{ll}\text{proj}(\text{flush}(\text{MA-stepn}(MA_0, n))) & \\ = \text{proj}(\text{flush}(\text{MA-step}(MA_{n-1}))) & \{\text{Def.}\} \\ = \text{ISA-stepn}(\text{proj}(\text{flush}(MA_{n-1})), k_{n-1}) & \{\text{A.1}\} \\ = \text{ISA-stepn}(\text{ISA-stepn}(\text{proj}(MA_0), \sum_{i=0}^{n-2} k_i), k_{n-1}) & \{\text{I.H.}\} \\ = \text{ISA-stepn}(\text{proj}(MA_0), \sum_{i=0}^{n-1} k_i) & \{\text{Def.}\} \ \Box\end{array}$$

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:

$$\text{proj}(\text{flush}(\text{MA-stepn}(MA_0, n))) = \text{ISA-stepn}(\text{proj}(MA_0), \mathrm{N}(MA_0, n)) \ .$$

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*

$$\text{proj}(\text{flush}(\text{MA-stepn}(MA_0, i))) = \text{ISA-stepn}(\text{proj}(MA_0), \mathrm{N}(MA_, i)) \tag{A.2}$$

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:*

$$\begin{aligned} &\text{proj}(\text{flush}(\text{MA-step}(MA_n))) \\ &= \text{ISA-stepn}(\text{proj}(\text{flush}(MA_n)), \mathrm{N}(MA_n, n+1) - \mathrm{N}(MA_n, n)) \end{aligned} \tag{A.3}$$

where $MA_n = \text{MA-stepn}(MA_0, n)$.

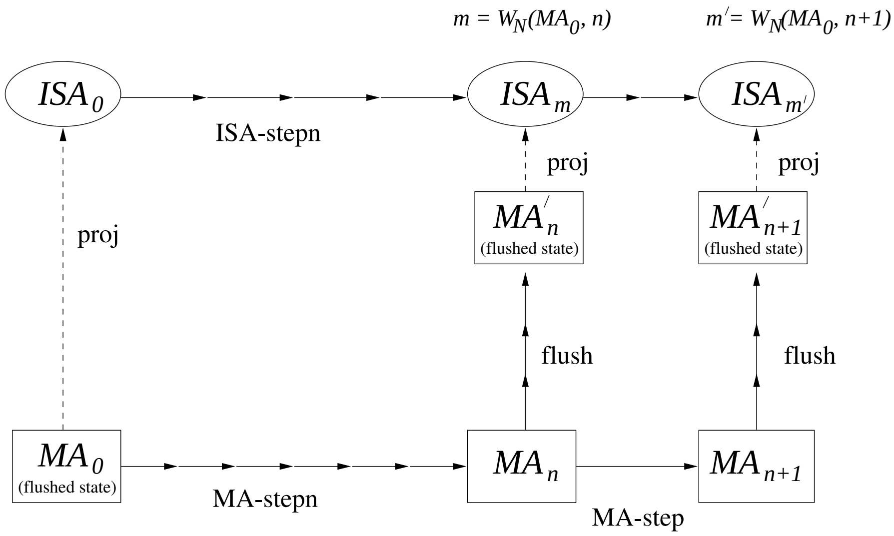

Figure A.1: Pictorial Proof of Theorem 6

Proof:

$$
\begin{aligned}
&\text{proj}(\text{flush}(\text{MA-step}(MA_n))) & \\
&= \text{proj}(\text{flush}(\text{MA-stepn}(MA_0, n+1))) & \{\text{Def.}\} \\
&= \text{ISA-stepn}(\text{proj}(MA_0), \text{N}(MA_0, n+1)) & \{(\text{A.2}),\ \text{i} = \text{n+1}\} \\
&= \text{ISA-stepn}(\text{ISA-stepn}(\text{proj}(MA_0), \text{N}(MA_0, n)), & \\
&\qquad\qquad \text{N}(MA_0, n+1) - \text{N}(MA_0, n)) & \{\text{Def. of ISA-stepn}\} \\
&= \text{ISA-stepn}(\text{proj}(\text{flush}(\text{MA-stepn}(MA_0, n))), & \\
&\qquad\qquad \text{N}(MA_0, n+1) - \text{N}(MA_0, n)) & \{(\text{A.2}),\ \text{i} = \text{n}\} \\
&= \text{ISA-stepn}(\text{proj}(\text{flush}(MA_n)), & \\
&\qquad\qquad \text{N}(MA_0, n+1) - \text{N}(MA_0, n)) & \{\text{Def.}\}\ \Box
\end{aligned}
$$

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

Due to the large volume of the verification scripts, they are not included in this compact version of the dissertation. All ACL2 proof scripts for the FM9801 are available at:

`http://www.utexas.edu/users/sawada/fm9801` or

`http://www.utexas.edu/users/sawada/fm9801/link.html` .

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.