Verification of a Pipelined Machine with Out-of-Order Instruction Completion *

Jun Sawada
Department of Computer Sciences,
The University of Texas,
Austin, TX 78712-1188, USA
E-mail: sawada@cs.utexas.edu

Warren A. Hunt, Jr.
Computational Logic, Inc.,
1717 West 6th St., Suite 290,
Austin, TX 78703-4795, USA
E-mail: hunt@cli.com

1 Project Goal and Overview

Our goal in this project is showing that the verification of complex pipelined machines is possible. As we discussed in the previous sections, the micro-architectural designs of general purpose microprocessors have not been thoroughly studied as a target of formal verification. It is not clear even how to represent the correctness of the super-scalar super-pipelined machines. Therefore, it is important to discuss what sort of correctness we can represent for processor designs, not only how we can verify it. So it is desirable not to restrict ourselves to a specific processor design. However, talking about processor designs in general is sometimes difficult and vague. In this paper, we would rather define a concrete pipelined machine model and discuss its verification. We believe that our method is not limited to our specific machine design, and we hope we can reuse our methodology for other processor designs in future projects.

Our pipelined machine model tries to capture several interesting features of today's pipelined microprocessor, such as out-of-order execution of instructions, speculative execution and multiple execution units. One of the hardest parts to verify in processor designs is the control logic, so we added a relatively complex issuing logic to our machine design, which stalls the pipeline whenever necessary to prevent data and structural hazards. We want to claim that the verification of our pipelined machine is much more difficult than a simple DLX pipeline implemation which has little pipeline interlocks, because it is this complexity in

*This research was supported in part by the Semiconductor Research Corporation under contract 96-DP-388.

the control logic that increases the number of the pipelined machine states that we have consider.

We have specified the machine at two levels, namely at the *ISA(instruction set architecture)* level and the micro-architectural level. The ISA specification is given as a function that takes the current ISA state and returns the next ISA state. In a single ISA transition, our ISA specification function executes exactly one instruction. In this sense, ISA specificaiton specifies the effect of individual instructions.

On the other hand, the micro-architectural level description is an abstract representation of an actual machine implementation. The micro-architectural description is given as a fuction that takes a micro-architectural state (or we simply call a *micro-state*) and returns the micro-state one clock cycle later. The micro-architectural description exposes the pipeline and other implementation details of the target machine design, by specifying how each micro-architectural component behaves in each clock cycle. However, it is still an abstract transition function which does not capture implementation details as much as an RTL description usually does. Note that a micro-state transition of a pipelined machine does not usually execute instructions one at a time, not like an ISA specification. It usually executes multiple instructions in the pipeline simultaneously, and takes several cycles to execute a whole instruction.

Our ISA and micro-architectural specifications are given as functions in the ACL2 logic. We also supply correctness criteria which can be viewed as the equivalence between the ISA and micro-architectural specification functions. We proved this correctness criteria with the ACL2 theorem prover, following the techniques we describe in the following chapters.

ACL2 definitions and theorems are usually written in files called *books*. Figure 1 shows the hierarchy of the ACL2 books constructed in our project. If an ACL2 book *include* other books with `include-book` command, it can access to the definitions and theorems in the included books. In the figure, a book from which an arrow starts includes the book pointed by the same arrow.

`ISA-def.lisp` defines our ISA specification, while `micro-archi-def.lisp` describes the micro-architectural design. Most of the books directly or indirectly include these specificaiton books, and define theorems about our machine. The final correctness theorem is proven in `correctness.lisp`. Books `liveness.lisp` and `diagram.lisp` also contain final results such as deadlock free property of the machine. We will discuss the functions and theorems defined in these books in the following sections.

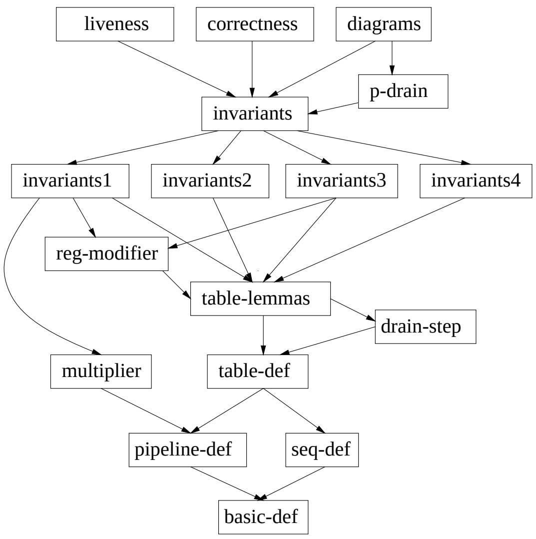

Figure 1: Major ACL2 books constructed in our project. Suffix ".lisp" is not shown.

2 Formal Specifications of a Pipelined Machine

2.1 Machine Specification with the IHS Library

We wrote our hardware specification using the *Integer Hardware Specification(IHS)*. IHS is based on the IHS library, which is an ACL2 library written by Bishop Brock. This subsection discuss the IHS library and the advantage in using IHS.

In theorem prover-based verification projects, a single data bit is often represented with a Boolean value, and a bit vector is by a list of Boolean values. For instance in the FM9001[7] specification, the Boolean true value T and false value F of the Nqthm logic[2] represent bit values 1 and 0, respectively. Four-bit bit vector 0101_2 is represented with a list of Boolean values (T F T F), where the least significant bit is the first element of the list.

Contrarily, the IHS library represents machine data such as bits and bit vectors using non-negative integers. For instance, bit values 1 and 0 are represented by integer 1 and 0. The bit vector 0101_2 is represented by integer 5.

One advantage of using IHS is the ease of specification. For example, the output of an adder taking input X and Y is represented by an arithmetic addition $X+Y$ in IHS. Similarly, other arithmetic operations can be specified easily, making the description of data-path abstract. Logical operations between bits and bit vectors are no less difficult than conventional representation, because many logical operations are defined in the IHS library. Examples of such logical operations defined in the IHS library are B-AND(a, b), LOGAND(X, Y) and

RDB(X, n, m), which represent logical-AND of bits a and b, bitwise-AND of bit vectors X and Y, and extraction of the n-to-m bit field from bit vector X, respectively. IHS also provides ACL2 macro DEFWORD, which helps us to define packed integer data structures such as instruction formats.

Another major advantage of using IHS is efficient execution of specifications. We can usually execute a machine specification written in the ACL2 logic, thanks to the execution capability of the ACL2 theorem prover. However, the speed of the execution heavily depends on the way the specification is written. Generally, execution of a specification using IHS is much faster than convectional specification using Boolean lists. For instance, concatenation of bit vectors X and Y in IHS can be defined as a simple arithmetic operation $X + 2^n \times Y$, where n is the width of the bit vector represented by X. On the other hand, the concatenation of bit vectors represented by Boolean lists requires a recursive function to append lists, which is more expensive than arithmetic operations.

One of the problems of IHS is its ambiguity. For instance, if we represent an adder with an arithmetic addition, it is not clear what type of adder is intended, either a ripple-carry adder or a shift-select adder. However, our interest of verification is the correctness of micro-architectural designs, not the correctness of small components like an adder. So the ambiguity of IHS in specifying micro-architectural components usually does not matter. Let us also note that we can still use IHS logical operations to define a specific component implementation, if we really want to.

There is also another ambiguity in IHS bit-vector specification; the integer representation of a bit vector does not specify the width of the vector. Integer 5 can represent a six-bit bit vector 000101_2 as well as a four-bit vector 0101_2. The user of the IHS libray has to pay attention to this ambiguity.

The IHS library contains a number of useful lemmas to help reasoning about the integer representation of bit vectors. These lemmas can be divided into three categories; math lemmas, quotient-remainder lemmas and lemmas about logical operators. Following three lemmas 1, 2 and 3 are examples from each category:

$$(0 < 1/x) \iff (0 < x) \tag{1}$$

$$\mathrm{REM}(x, y) = x \ \text{ when } |x| < |y| \tag{2}$$

$$\mathrm{LOGXOR}(i, j) = \mathrm{LOGXOR}(j, i) \tag{3}$$

The IHS library is included by all the books constructed in our project, even though Fig 1 does not show it. During the project, we found that additional logical operators and lemmas are useful for our verification task. We defined the logical operators in ACL2 book `b-ops-aux-def.lisp` and the lemmas in `b-ops-aux.lisp`. These books are also included by most of the ACL2 books in Fig 1, but again they are not shown in the figure. The lemmas in the IHS library and the augmenting books are automatically referenced by the theorem prover when they are necessary.

2.2 Basic Definitions of Machine Components

A microprocessor design usually has a number of registers and state holding devices in it. In order to specify the state of a complete processor, we have to specify the state of each device. One of the simplest state holding devices in microprocessor designs may be a program counter. Since a program counter contains only the next instruction address, we can represent its state with a single non-negative integer using IHS. More complex state holding devices, such as a register file, pipeline latches and memory, need complex data structures.

For example, let us discuss how we represent a register file. A register file with n registers can be represented with a list of n non-negative integers, where i'th integer in the list represents the bit vector held by register Ri. For example, list of integers (0 1 2 3 ...) represents a state of a register file whose R0 register contains value 0, R1 register contains 1, and so on. Using ACL2 function $\mathrm{nth}(i, l)$ which returns the i'th element of list l, the read access to the i'th register in register file *regs* can be defined as:

$$\text{read-reg}(i, \mathit{regs}) = \mathrm{nth}(i, \mathit{regs}). \tag{4}$$

Similarly, the function write-reg which returns the resulting register file state after storing value *val* in the i'th register of register file *regs* can be defined as:

$$\text{write-reg}(\mathit{val}, i, \mathit{regs}) = \text{update-nth}(i, \mathit{val}, \mathit{regs}), \tag{5}$$

where $\text{update-nth}(i, val, l)$ is an ACL2 function that replaces the i'th elememnt of list l with value *val*.

From the definition of read-reg and write-reg, we can prove basic properties about register accesses. Following lemma is one of them:

Lemma 1 (read-write-reg) Suppose *rn1* and *rn2* are register identifiers and *regs* is a register file, then:

$$\text{read-reg}(\mathit{rn1}, \text{write-reg}(\mathit{val}, \mathit{rn2}, \mathit{regs})) = \mathrm{IF}(\mathit{rn1} = \mathit{rn2}, \mathit{val}, \text{read-reg}(\mathit{rn1}, \mathit{regs})).$$

The lemma says that a read access to a modified register returns the new value, and the values in the rest of the registers are unchanged. Figure 2.2 shows the ACL2 defintions of read-reg write-reg, and Lemma 1.

Similarly, memory can be defined using a list of non-negative integers, and properties about the behavior of memory can be proven. One major difference from the register file definition is that we split the memory space into half, and made the first half of memory read-only, while the other half writable. We allow fetching instructions only from the read-only portion of memory. This eliminates the problem of self-modifying code. Pipelined machine with self-modifying code is hard to design, and many implementation of microprocessor does not guarantee the correct operation with such code.

```
(defun read-reg (i regs) (nth i regs))

(defun write-reg (val i regs)  (update-nth i val regs))

(defthm read-reg-write-reg
    (implies (and (rname-p rn1)
                  (rname-p rn2)
                  (regs-p regs))
             (equal (read-reg rn1 (write-reg val rn2 regs))
                    (if (equal rn1 rn2) val (read-reg rn1 regs)))))
```

Figure 2: Definition of read-reg, write-reg and Lemma 1

The list representation of memory is actually very inefficitient in the execution of the specification, because list operation is often linear to the length of the list. Using binary tree is probably a better representation for its execution. However, it does not make much difference in the proof of machine design correctness. Once important properties of memory and register files behavior are proved, we can forget the actual definition and prove the correctness of the complete machine using these properties.

The definition of a register file and memory is given in basic-def.lisp.

2.3 ISA Specification

We give two different specifications of our example machine. One is ISA specification and the other is micro-architectural specification. ISA specification specification formally defines the effect of individual instructions on the machine state visible from a programmer. However, ISA specification addresses none of the implementation issues of the pipelined machine. Micro-architectural specification is the most abstract description that exposes such implementation issues.

We designed our ISA simplest as possible. It has only a program counter, a register file and memory. No conditional flags and special registers exist. The register file contains 16 registers, each of which can hold a 32-bit word. The memory has a space of $2^16 = 65536$ addresses, and a 32-bit word is stored at each address, just like the FM9001 design. As briefly discussed in the previous subsection, the first half of the memory space is read-only, and the processor can fetch its instructions only from this space. The program counter contains only 15-bit address, and can only point to the first half of the space.

We represent an ISA state with an ACL2 structure (ISA *pc regs mem*)[1] , where *pc* is an integer-based representation of 15-bit bit vector, and *regs* and

[1]The pure ACL2 logic does not have a data-structure corresponding to Common Lisp "structure". Instead, we use ACL2 structure library. Using this library, the structure with the name "STRUCT1" is represented by a list starting with the symbol "STRUCT1".

Opcode	Effect on ISA
ADD Rc,Ra,Rb	$Regs[Rc] \leftarrow Regs(Ra + Rb)$
MULT Rc,Ra,Rb	$Regs[Rc] \leftarrow Regs[Ra] * Regs[Rb]$
BZ Rc,Ra,Rb	$PC \leftarrow Regs[Ra] + Regs[Rb]$ if $Rc = 0$
LD Rc,Ra,Rb	$Regs[Rc] \leftarrow Mem[Regs[Ra] + Regs[Rb]]$
ST Rc,Ra,Rb	$Mem[Regs[Ra] + Regs[Rb]] \leftarrow Regs[Rc]$
SYNC	No effect on ISA.

Table 1: List of Instructions.

```
(defun ISA-step (s)
  (let ((inst (read-mem (addr-word (ISA-state-pc s)) (ISA-state-mem s))))
    (let ((op (opcode inst))
          (rc (rc-field inst))
          (ra (ra-field inst))
          (rb (rb-field inst)))
      (cond ((equal op 0)                ; add
             (ISA-add rc ra rb s))
            ((equal op 1)                ; mult
             (ISA-mult rc ra rb s))
            ((equal op 2)                ; branch
             (ISA-branch rc ra rb s))
            ((equal op 3)                ; load
             (ISA-load rc ra rb s))
            ((equal op 4)                ; store
             (ISA-store rc ra rb s))
            ((equal op 5)                ; sync
             (ISA-sync s))
            (t (ISA-undefined s))))))
```

mem represents the state of the register file and the memory in the way discussed in the previous subsection.

Our example machine implements only six instructions, each of which corresponding to an instruction family of an ordinary processor. The list of the instructions are given in Table???. The SYNC instruction does not have any effect on ISA state. It is an instruction which flushs all the preceding instructions out of pipeline. Since our example machine does not implement interrupts, we treat an undefined instructions as an NOOP.

The behavior of these instructions are formally specified by defining a next ISA-state function in the ACL2 logic. A next ISA-state function takes a ISA state and returns the ISA state after executing one instruction.

The definition of the next ISA-state function, ISA-step, is as shown bellow: The second line of the definition sets the local variable `inst` to the instruction pointed by the program counter in the ISA state `s`. The third line extracts the field of opcode, Ra, Rb and Rc. Then, depending on the opcode, the function calls an appropriate function which defines the effect of the instruction represented by the opcode. For instance, ADD and MULT instructions are defined as:

```
(defun ISA-add (rc ra rb s)
  (let ((regs (ISA-state-regs s)))
    (ISA-state (pc-word (1+ (ISA-state-pc s)))
               (write-reg (word (+ (read-reg ra regs) (read-reg rb regs)))
                          rc regs)
               (ISA-state-mem s))))

(defun ISA-mult (rc ra rb s)
  (let ((regs (ISA-state-regs s)))
    (ISA-state (pc-word (1+ (ISA-state-pc s)))
               (write-reg (word (* (read-reg ra regs) (read-reg rb regs)))
                          rc regs)
               (ISA-state-mem s))))
```

These instructions update the register specified by Rc with the sum or product of the register values of Ra and Rb, and increment the program counter. The ST instruction is defined as:

```
(defun ISA-store (rc ra rb s)
  (let ((regs (ISA-state-regs s)))
    (ISA-state (pc-word (1+ (ISA-state-pc s)))
               (ISA-state-regs s)
               (write-mem (read-reg rc regs)
                          (addr-word (+ (read-reg ra regs) (read-reg rb regs)))
                          (ISA-state-mem s)))))
```

This definition updates the memory at the correct address with the value of register Rc.

In this way, the effect of individual instructions are clearly specified in the ISA specification. On the other hand, the implementation and behavior of the pipelined machine is not visible in the specification.

2.4 Micro-Architectural Specification

The micro-architectural specification describes the implementation issues such as pipelining, which are not specified in the ISA-specification. However, it is still an abstract behavioral description of hardware, and does not specify the implementation detail. More detailed descriptions is usually given by an RTL, but we don't discuss it here.

The block diagram of our example pipelined machine is given in Figure ??. The machine is divided into several stages by pipeline latches, which are shown by patterned rectangles in the figure. Generally speaking, the pipeline is divided into the fetch, decode, execution and write-back stages. However, the actual number of stages the executed instruction go through in the pipeline varies from four to six depending on the instruction.

Table??? shows how each instruction advances through the pipeline. For instance, an ADD instruction takes only one cycle to be executed. An decoded

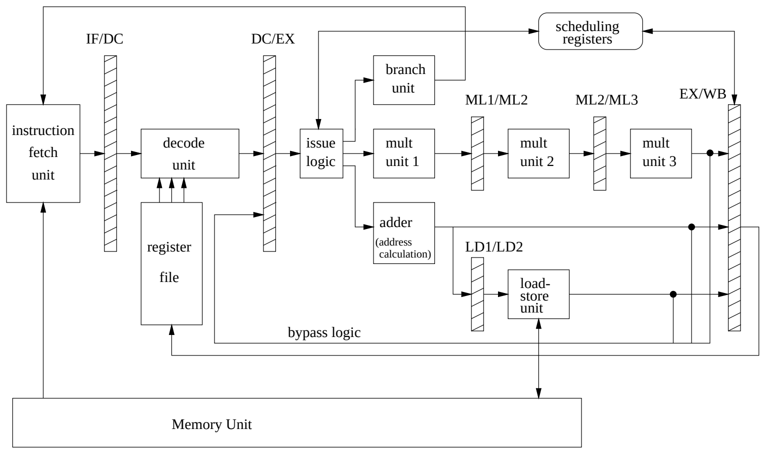

Figure 3: Block Diagram of an example pipelined machine.

ADD instruction is issued from the DC/EX latch to the adder, and is directed to the latch EX/WB. In the next cycle, its result is written back into the register file.

ADD	IF/DC → DC/EX → EX/WB
LD	IF/DC → DC/EX → LD1/LD2 → EX/WB
MULT	IF/DC → DC/EX → ML1/ML2 → ML2/ML3 → EX/WB
ST	IF/DC → DC/EX → LD1/LD2
BZ	IF/DC → DC/EX
SYNC	IF/DC → DC/EX

The memory access instruction takes more cycles to be executed. In the first cycle after the issue of an instruction LD or ST, it is sent to the adder to calculate the access address and then stored in LD1/LD2 latch. In the next cycle, the actual memory access takes place. Since we assume that the memory responds always, the memory access instructions takes exactly two cycles.

The multiplication takes three cycles to be executed, going through latches ML1/ML2 and ML2/ML3. The intention of introducing such a pipelined multiplier is modeling the floating point unit, which usually takes more cycles than an integer unit. The multiplier is defined using a Wallace tree.

Since different instructions take different number of cycles to be executed, this pipelined machine may complete instructions out-of-order, even if instructions are issued in-order. Especially, if an ADD instruction is issued one cycle after an MULT instruction, the ADD instruction completes earlier than the MULT instruction. There is no re-order buffer in our machine design, so the instructions may write back their results to the register file out-of-order.

In order to make the pipelined machine work correctly, we have to avoid various pipeline hazards. One of the possible hazards is read-after-write, or RAW, hazards[?]. Suppose we execute instruction I and then a following instruction J. If instruction J reads the register updated by instruction I, J has to wait until I writeback its result. So the pipeline has to stall the execution of J for appropriate number of clock cycles. Actually, our pipeline has a bypass logic to reduce this stall cycles, but it does not completely eliminate the need for the stalls.

Write-after-write, or WAW, hazard is another possibility. If instructions I and J completes and write back their results out-of-order, and their destination registers are the same, the resulting register file may not store the result of the correct computation. We have to make sure no out-of-order writebacks happen between instruction with the same destination register.

We also have to worry about structural hazards. One of the possible structural hazard can occur at the writeback stage. Because different instructions take different number of execution cycles, two instructions may try to enter the writeback stage simultaneously. We have to arbitrate or pre-schedule conflicting instructions to avoid such hazards.

Our pipelined machine resolves the hazards described above by schduling the issue of instructions dynamically. The pipeline latch DC/EX works like a *reservation station*. An instruction is stored into the DC/EX latch as soon as it is decoded, even if the source register value is still calculated by a preceding instruction in the pipeline. The instruction waits in DC/EX until all the source register value is forwarded through the bypass logic. The instruction also have to wait if there is a possibility of WAW hazards or structural hazards at the writeback stage. Once an instruction is issued to the execution unit, it is guaranteed that no hazard occurs with respect to the instruction. So an issued instruction never stalls in our current design.

Since the issuing logic has to consider all possible cases of instruction conflicts, it is one of the most complicated part of our design. In order to make the issuing logic less complex, we added scheduling registers which keep track of the instructions in the execution unit[2]. One of the major problem in our verification task is verifying that this hazard-preventing mechanizm work correctly.

Our machine also has a simple mechanism of speculative fetching on conditional branches. If a conditional branch instruction, BZ, is fetched by the pipeline, we assume that the branch is not taken and continue fetching the instruction from the succeeding address. The branch is resolved at the third stage of the pipeline, so we may possibly load two speculative instructions. If the branch turned out to be actually taken, then we abandon all the speculatively fetched instructions, and begin to fetch instructions from the target address of the branch.

There are two thing to note about speculative fetching. Since the instruction

[2]Scheduling registers are also refered to by the logic directing the output of execution unit to the EX/WB latch.

issues are in-order, and the branch condition is resolved immediately after the issue, no speculative instructions will be issued to the execution unit. The program counter contains the correct address of the next instruction if and only if no conditional branch instruction, which actually takes the branch, is in the pipeline.

2.5 Micro-architectural Specification

The micro-architectural specification specifies the implementation detail discussed in the previous section. It describes how each micro-architectural component behave in each clock cycle. It is still a behavioral specification, and may look quite different from realistic hardware description like RTL.

A micro-architectural design usually contains state holding devices which are for implementation purposes and not visible to the programers. The state of these devices, which is sometimes called implementation registers, should be included in a machine state at micro-architectural level. In our pipelined machine design, pipeline latches and scheduling registers are implementation registers. So the following ACL2 structure is used to represent a micro-architectural state, or simply a micro-state:

```
(micro-state Pc Regs Mem Lch Schedlr),
```

where Lch is a structure representing all the pipeline latches and Schedlr represents the state of scheduling registers.

As we did in defining the ISA-specification, we define a next micro-state function `micro-state-step`. `Micro-state-step` takes a micro-state of the machine as an argument and returns the micro-state after one clock cycle of execution. Note that next state function at ISA level returns the state after execution one instruction, while the next state function at micro-architectural level returns a state after one clock cycle.

The definition of `micro-state-step` is as follows:

```
(defun micro-state-step (s)
  (micro-state (step-pc s)
               (step-regs s)
               (step-mem s)
               (step-lch s)
               (step-schedlr s)))
```

The step functions used in the definition describe how micro-architectural components are updated in this clock cycle. For instance, `step-pc` describes how the program counter is updated after one cycle of execution. `Step-lch` defines the way the pipeline latches are updated as:

```
(defun step-lch (s)
  (lch (step-IF/DC s)
```

```
        (step-DC/EX s)
        (step-EX/WB s)
        (step-LD1/LD2 s)
        (step-ML1/ML2 s)
        (step-ML2/ML3 s))),
```

where `step-IF/DC` defines the next state of IF/DC latch, `step-DC/EX` defines that of DC/EX latch, and so on. Compared with the ISA-specification, which describes the machine behavior with respect to individual instruction, the micro-architectural specification pay more attentions to the componentwise behavior of the pipelined machine design.

In Figure???, we put a partial definition of step-DC/EX. The next state of DC/EX latch when a pipeline stall occurs at the latch is described byt the function `step-waiting-DC/EX`. `Bypass-update-in-wait` describes how each register value field in DC/EX is updated through the bypass logic.

```
(defun bypass-update-in-wait (val rname s)
  (let ((schedlr (micro-state-schedlr s)))
    (let ((shift1 (schedlr-shift1 schedlr)))
      (b-if (bs-and (shift-reserved? shift1)
                    (bv-eqv *rname-size* (shift-rname shift1) rname)
                    (bv-eqv 2 (shift-exunit (schedlr-shift1 schedlr)) 2))
            (ML-output s)
            (b-if (bs-and (shift-reserved? shift1)
                          (bv-eqv *rname-size* (shift-rname shift1) rname)
                          (bv-eqv 2 (shift-exunit (schedlr-shift1 schedlr)) 3))
                  (LD-output s)
                  val)))))

; A state of DC/EX is represented by a struction of form
;(DC/EX valid-bit pc control-vector rc-value ra-value rb-value
;       rc-name rb-name rc-name)
;
(defun step-waiting-DC/EX (s)
  (let ((latch (lch-DC/EX (micro-state-lch s))))
    (DC/EX (DC/EX-valid latch)
           (DC/EX-pc latch)
           (DC/EX-cntlv latch)
           (bypass-update-in-wait (DC/EX-rc latch) (DC/EX-rcn latch) s)
           (bypass-update-in-wait (DC/EX-ra latch) (DC/EX-ran latch) s)
           (bypass-update-in-wait (DC/EX-rb latch) (DC/EX-rbn latch) s)
           (DC/EX-rcn latch)
           (DC/EX-ran latch)
           (DC/EX-rbn latch))))
```

```
(defun step-DC/EX (s)
  (b-if (b-ior (branch-taken? s)
               (release-sync? s))
        (step-reading-DC/EX 0 s)
        (b-if (DC/EX-stall? s)
              (step-waiting-DC/EX s)
              (step-reading-DC/EX 1 s))))
```

2.6 Micro Architectural Execution Trace Table

Our approach incrementally proves various pipeline properties of our machine design, and then use those proven propreties to verify the correctness criteria. So our first task is defining various invariant properties of the pipeline designs. However, we found that it is difficult to write some pipeline properties as a predicate taking the micro-state as its sole argument. It is not easy to give a single precise reason for the difficulty, but we can point out several possible reasons for that.

First, some pipeline invariants are easier to define over the instructions rather than the machine state. For instance, RAW-hazard is closely related to RAW dependency between instructions. If instruction I is RAW-dependent on instruction J, it is so in the next clock cycle. However, the invariantness of RAW-dependency between instructions is not easy to define as a predicate taking a micro-state. We have to specify, for instance, if the instruction at latch IF/DC is RAW-dependent on the instruction at latch DC/EX, the instruction at DC/EX at the next cycle is RAW-depedent on the instruction at latch EX/WB, and so on.

Second, since the micro-architectural designs are usually optimized for speed, a micro-state may not contain enough information to directly define pipeline properties. For instance, consider defining a predicate determining RAW-dependency between instructions in our pipelined machine. Express the dependency between instructions at latches IF/DC and DC/EX are easy. The following expression determines if the destination register RC of the instruction at DC/EX is equal to the source register RA of the instruction at IF/DC.

```
(EQUAL (IF/DC-ran (lch-DC/EX (micro-state-lch s))
       (DC/EX-rcn (lch-DC/EX (micro-state-lch s))
```

However, similar equalities are difficult to write for instructions which are already issued, because the source register name is abandoned as soon as instructions are issued.

Another problem is that writing these invariant properties requires the knowledge of precise timing of pipeline events. Like the skewed abstraction mapping discussed in the previous chapter, such invariants become increasingly difficult to define when pipeline stalls and complex control features are introduced to the machine.

It is our believe that we need a new representation of micro-atate to ease defining invariant properties. We reasoned that an instruction-based information-rich representation will help us to define certain invariants like dependencies between instructions. The Micro-Architectural Execution Trace Table(MAETT) is the new representation we will use here.

A MAETT is a trace table of instructions which are currently being executed or has already been retired. A MAETT is defined to be a list of table entries, each of which represents a specific instruction. MAETT entries are stored in such a way that the represented instructions are in the ISA-execution order. A MAETT may contain multiple entries which represent the same instruction at the same address if it is executed more than once.

Each MAETT entry stores the current status of the represented instruction. The stored information includes the instruction word, the instruction address, source and destination registers. The complete list of stored information is in Table???. Since MAETT entries record rich information related to all the instruction in the pipeline, we can reconstruct the current micro-state corresponding to a MAETT. In this sense, a MAETT is another representation of a micro-state.

Since a MAETT contains a rich information about instructions, it is easy to define concepts related to instructions. For instance, instructions I and J are represented by MAETT entries E_I and E_J. The predicate defining the RAW-dependency of J on I can be written as:

```
(OR (EQUAL (ENTRY-RCN E_I) (ENTRY-RAN E_J))
    (EQUAL (ENTRY-RCN E_I) (ENTRY-RBN E_J)))
```

In other words, the destination register Rc of instruction I is a source register Ra or Rb of instruction J, then instruction J RAW-depends on I.

The MAETT representation has another trick for defining pipeline property by storing register file and memory image as shown in Table???. Let us consider the sequential ISA execution of instructions $I_1, I_2, \ldots, I_n$. From programmer's viewpoint, there exists an ISA-state S_0 before executing instruction I_1, and the machine state transits to a new state S_1. In the next cycle of ISA-execution, instruction I_2 is executed and the machine state transits from S_1 to another new state S_2. In this way, we can define ISA-state transition sequence $S_0, \ldots, S_n$ which satisfy the relationship:

$$S_{i+1} = ISA - step(S_i)$$

The state S_i is the "ideal" ISA-state right after executing instruction I_{i-1} and before executing I_i. We store the register file and memory of S_i to the MAETT entry E_i representing instruction I_i. Since the program counter stored in E_i points to I_i itself, the "ideal" state S_i can be represented as:

$$S_i = (ISA - state(entry - pcE_i)(entry - regsE_i)(entry - memE_i))$$

where entry-pc, entry-regs and entry-mem are the accessors to the corresponding record in a MAETT entry.

Using the "ideal" state of register file and memory, certain properties are easy to define. For instance, we may want to represent the correct source register value for an issued instruction. The value of the register in the current micro-state is certainly not the right value, if a previously issued instruction is to update the register. Using the "ideal" register file image, we can represent the correct value of the source register Ra as:

```
(read-reg (rnames-ran (entry-rnames E))
          (entry-regs E)))))
```

where (rnames-ran (entry-rnames entry)) specifies the Ra operand register for the instruction represented by the MAETT entry E.

Let us note that considering "ideal" ISA-execution of instructions implies that there may not be a corresponding MAETT representation for arbitrary micro-states. Only if a micro-state is a reachable state from a flushed pipeline state, we can naturally define the "ideal" ISA-state at each step of ISA-execution. We will define this validity of MAETT later.

Since a MAETT is another representation of a micro-state, we can also define a MAETT updating function `MAETT-step`, which simulates the micro-state transition `micro-state-step`. Since the MAETT is a list of MAETT entries, `MAETT-step` is defined to update each MAETT entry depending on how the corresponding instruction behave in the pipeline. `MAETT-step` also addes an MAETT entry corresponding to a newly fetched instruction, or delete entries corresponding to speculatively executed instruction, if necessary.

Actual definition of `MAETT-step` has a slight technical problem. Since instructions in the pipeline interfere with each other, we need to look up the whole machine state to determine how each instruction behave in the pipeline. So the function `step-entry` which updates each MAETT entry is defined to be a function which takes a MAETT entry `e` and the corresponding micro-state `s` as arguments:

```
(defun step-entry (e s)
  (case (entry-stg e)
    (IF/DC (step-e-IF/DC e s))
    (DC/EX (step-e-DC/EX e s))
    (EX/WB (step-e-EX/WB e s))
    (LD1/LD2 (step-e-LD1/LD2 e s))
    (ML1/ML2 (step-e-ML1/ML2 e s))
    (ML2/ML3 (step-e-ML2/ML3 e s))
    (retire e)
    (otherwise nil)))
```

`step-entry` dispatches the definition depending on the current stage of the instruction represented by `e`. For example, `step-e-IF/DC` defines the update of MAETT entry representing an instruction at IF/DC latch:

```
(defun step-e-IF/DC (e s)
  (update-entry e
        :stg (b-if (IF/DC-stall? s) *IF/DC* *DC/EX*)
        :misc
        (b-if (IF/DC-stall? s)
              (entry-misc e)
              (misc-DC/EX (decode (entry-inst e))
                          (read-bypassed-reg (rnames-rcn (entry-rnames e))
                              (step-regs s) s)
                          (read-bypassed-reg (rnames-ran (entry-rnames e))
                              (step-regs s) s)
                          (read-bypassed-reg (rnames-rbn (entry-rnames e))
                                             (step-regs s) s)))))
```

The next state of IF/DC, which is specified after `:stg`, depends on the value of the control logic `IF/DC-stall?` in the micro-architectural definition.

2.7 Invariant Pipeline Properties

There are two major reasons to define and reason about invariant properties in our verification project. The first reason is for dividing the verification problem. We can prove the correctness of the machine design incrementally by proving the invariant properties one by one and then the correctness criteria.

The other reason is restricting machine states to reachable machine states so that we have to consider only reasonable machine states. This is essential especially when we consider Burch and Dill's Diagram (a) in Figure ??. As is discussed in [?], Daagram (a) does not hold for arbitrary "old micro-state", but only for a certain subset of micro-states which is constrained by invariant conditions.

There are a number of properties which are invariantly held by our micro-architectural design. We will first explain the definition of a few example properties to show how we defined them with the MAETT representation.

First example is about structural hazards. A structural hazard is a resource conflict where more than one instruction demands the use of the same hardware component. One of the possible structural hazard in our design is that multiple instruction may try to occupy the same pipeline latch simultaneously. It is not trivial that our pipeline design does not have such hazards, because the execution of instructions can take different number of cycles, which may cause multiple instruction to simultaneously advance to the latch EX/WB before the writeback stage.

In the ACL2 logic, the hazard-free property `no-structureal-hazard?` is defined as a recursive function taking a MAETT `tbl` as its sole argument:

```
(defun member-entry-stg (stg tbl)
  (if (endp tbl)
      nil
      (if (equal stg (entry-stg (car tbl)))
  tbl
  (member-entry-stg stg (cdr tbl)))))


(defun no-structural-hazard? (tbl)
  (if (endp tbl)
      T
      (if (equal (entry-stg (car tbl)) *retire*)
  (no-structural-hazard? (cdr tbl))
  (and (not (member-entry-stg (entry-stg (car tbl)) (cdr tbl)))
       (no-structural-hazard? (cdr tbl))))))
```

`No-structural-harzard` recursively checks whether no pair of MAETT entries E_1 and E_2 satisfy the following relation:

$$\texttt{(entry-stg } E_1) = (\text{entry-stg } E_2) \neq \text{*RETIRE*}$$

Another example is about WAW-hazards. The hazard-free property for WAW-hazard is defined by a recursive predicate `no-WAW-hazard?` as:

```
(defun rec-WAW-violation? (entry tbl)
  (if (endp tbl)
      nil
      (if (WAW-violation? entry (car tbl))
  T
  (rec-WAW-violation? entry (cdr tbl)))))

(defun no-WAW-hazard? (tbl)
  (if (endp tbl)
      T
      (and (not (rec-WAW-violation? (car tbl) (cdr tbl)))
   (no-WAW-hazard? (cdr tbl)))))
```

In other words, `No-WAW-hazard?` checks recursively whether no WAW-hazard occurs betwen instructions in MAETT `tbl`. The WAW-hazard between the instructions represented by MAETT entries *entry1* and *entry2* is defined as:

```
(defun WAW-violation? (entry1 entry2)
  (and (reg-writeback-inst? (entry-inst entry1))
```

```
        (reg-writeback-inst? (entry-inst entry2))
        (same-destination-reg? entry1 entry2)
        (out-of-order-retire? entry1 entry2)))
```

This definition assumes that *entry1* represents an instruction earlier than the one represented by *entry2*, with regard to the ISA-execution order . The definition says that a WAW-hazard between two instructions occurs when both of the instructions are write-back instructions, they share the same destination register, and instruction *entry2* retires earlier than instruction *entry1*. Each of the predicates in the definition is further defined as a relation of the fields of the MAETT entries. For example, the refinement of `out-of-order-retire?` is:

```
(defun out-of-order-retire-stages? (stg1 stg2)
  (declare (xargs :guard (and (stg-name-p stg1) (stg-name-p stg2))))
  (case stg1
    (EX/WB (member-equal stg2 (list *RETIRE*)))
    (LD1/LD2 (member-equal stg2 (list *EX/WB* *RETIRE*)))
    (ML2/ML3 (member-equal stg2 (list *EX/WB* *RETIRE*)))
    (ML1/ML2 (member-equal stg2 (list *LD1/LD2* *ML2/ML3* *EX/WB* *RETIRE*)))
    (otherwise nil)))
```

The first case of the definition tells that `out-of-order-retire?(`$entry1$`,`$entry2$`)` is true if the current stage *stg1* of an earlier instruction *entry1* is the writeback stage EX/WB when a following instruction *entry2* is already retired.

The last example property is `correct-regs?`, which says that the register file records the results of all the retired instructions. The question is how to define the correct state of register file at clock cycle. If the pipelined machine completes the instructions in order, any ISA-state has a corresponding micro-state whose register file is in the same as that of the ISA-state. However, our machine with out-of-order execution can updated the register file in an out-of-order fashion. So it is difficult to relate register file images in ISA-states and micro-states.

We define the correct register file state in a micro-architecture using the corresponding MAETT. Remember each MAETT entry records the state of "ideal" register file. So the register file image recorded in the first MAETT entry is the initial register file before executing any instruction. The correct current state of register file can be calculated by accumulating all the register updates by the retired instructions in a MAETT.

The ACL2 definition of correct-regs? is shown below. The correct state of register file is `(calc-current-regs tbl)`. `(Accumlt-regs-updates regs tbl)` is the recursive function which accumulates the register updates by all the retired instruction in `tbl`. `(entry-regs (car tbl))` is the initial state of register file.

```
(defun accumlt-regs-updates (regs tbl)
```

```
  (if (endp tbl)
      regs
      (accumlt-regs-updates (if (equal (entry-stg (car tbl)) *retire*)
(update-regs-by (car tbl) regs)
regs)
    (cdr tbl))))


(defun calc-current-regs (tbl)
  (accumlt-regs-updates (entry-regs (car tbl)) tbl))

(defun correct-regs? (tbl s)
  (or (endp tbl)
      (equal (micro-state-regs s)
     (calc-current-regs tbl))))
```

The property of correct-regs? implies that the register file in the actual micro-state `s` is equal to the correct register file calculated by `calc-current-regs`.

In a similar manner, we have used the MAETT representation to define other pipeline properties which are essential to the correct operation of our design:

`no-structural-hazard?` : No structural hazards occur.

`intermediate-values-ok?` : Functional correctness of the data-path. We defined this by checking that the intermediate values of the instruction at each stage are correct. For instance, an instruction at the issue stage should have obtained the correct source register values from either the register itself or the bypass logic. This property entails non-existence of RAW-hazards.

`all-insts-in-MAETT?` : All the partially executed instructions in the pipeline have corresponding entries in the MAETT.

`consistent-scheduler?` : The scheduling registers correctly keep track of instructions in the execution units.

`speculative-inst-not-issued?` : No speculatively fetched instructions are issued to the execution units.

`speculative-fetching-ok?` : All speculatively fetched instructions are abandoned before instructions are fetched from the correct branch.

`in-order-issue?` : Instruction issues are in-order.

`no-WAW-hazard?` : No WAW-hazards occur.

`in-order-memory-accs?` : No out-of-order memory accesses are performed.

`sync-retires-insts?` : All instructions retire before a following SYNC instruction is itself retired.

`consistent-ISA-states?` : The "ideal" ISA states corresponding to the instruction is stored in the MAETT. This is checked against the ISA specification.

`correct-pc?` : The program counter is pointing to the next instruction to be fetched.

`correct-regs?` : The register file is correct. The correct register file is defined by scanning the MAETT and sequentially accumulating all the side-effects by the retired instructions on the initial register file. In other words, the register file should record all the updates caused by the retired instructions.

`correct-mem?` : The memory should record all the updates by the retired STORE instructions in the MAETT.

We define the conjunction of all of these properties to be predicate `MAETT-ok?`. `MAETT-ok?` can be viewed as the well-defined predicate for a MAETT as well as an property over the micro-architectural design..

```
(defun MAETT-ok? (tbl s)
  (and (MAETT-p tbl)
       (no-structural-hazard? tbl)
       (intermdt-values-ok? tbl s)
       (all-insts-in-MAETT? tbl s)
       (consistent-scheduler? tbl s)
       (consistent-ISA-states? tbl)
       (speculative-inst-not-issued? tbl)
       (speculative-fetching-ok? tbl)
       (in-order-issue? tbl)
       (correct-regs? tbl s)
       (correct-mem? tbl s)
       (no-WAW-hazard? tbl)
       (in-order-memory-accs? tbl)
       (correct-pc? tbl s)
       (sync-inst-flush? tbl)))
```

2.8 Proof of Invariant Properties

The predicate `MAETT-ok?` defined in the previous subsection is invariantly held over the micro-state transitions. More concretely, we can prove a lemma:

Lemma 2 (Invariant property of MAETT-ok?) *Suppose s is a micro-architectural state and M is the corresponding $MAETT$ satisfying* `MAETT-ok?(`M`,`s`)`.

Then the next micro-architectural state `micro-state-step`(s) *and the updated MAETT* `MAETT-step`(M,s) *satisfy:*

$$\texttt{MAETT-ok?}(\texttt{MAETT-step}(M, s), \texttt{micro-state-step}(s)).$$

In addition to this, we can say that the empty MAETT `nil` is a valid MAETT representation of any flushed micro-state.

Lemma 3 (MAETT for a Flushed State) *Suppose s is a flushed micro-state; that is, no partially executed instruction is in the pipeline of s. The empty MAETT* `nil` *is a correct MAETT representing s, and* `MAETT-ok?`$(\texttt{nil}, s)$ *is true.*

From these two lemmas, all the micro-states reachable from a flushed pipeline state has a correct MAETT representation and `MAETT-ok?` is true. In other words, all the reachable micro-states satisfy the properties defined in the previous subsection.

The proof of Lemma 3 is simple, but the proof of Lemma 2 requires a profound analysis of the micro-architectural design. Actually, most of our verification efforts in our project were directed toward the proof of this lemma. Once we prove these two lemmas, the correctness criteria for our machine design can be fairly easily proved from them.

Since the `MAETT-ok?` is a conjunction of the properties given in the previous subsection, we can prove Lemma 2 by showing that each property is true with `micro-state-step`(s) and `MAETT-step`(M,s). Because individual properties are relevant to a small part of the complete micro-architectural design, we can concentrate our verification effort in the related area of the machine design. For instance, when we prove the property `no-WAW-hazards?`, we can concentrate our effort on the issuing logic and ignore, for instance, the branching unit. Then, assuming no WAW-hazard occurs, we can prove `correct-regs?` by concentrating our efforts on the writeback stage to the register file. In this way, the complete verification problem is divided into local problems, which are verified incrementally. This incremental verification approach helped us avoid case explosions in our proof and reduced the computational cost of the verification process.

In the rest of this subsection, we will discuss how we proved some of the MAETT-ok? conjuncts.

The first example is the property `no-structural-hazard?`. The lemma we prove is:

Lemma 4 *Suppose s is a micro-state and M is its MAETT representation satisfying MAETT-ok?(M,s). (This implies no-structural-hazard?(M).) Then the MAETT representation MAETT-step(M,s) representing the next micro-state micro-state-step(s) is satisfyies no-structural-hazard(MAETT-step(M,s)).*

Because predicate `no-structural-hazard?` is defined to check that no pair of MAETT entries occupy the same pipeline stages, we can prove Lemma 4 from the following sublemma.

Lemma 5 *Suppose s is a micro-state and E_1 and E_2 are MAETT entries corresponding to s. Further suppose E_1' and E_2' are updated MAETT entries satisfying E_1' = step-entry(E_1), E_2' = step-entry(E_2). If the current stages entry-stg(E_1) and entry-stg(E_2) of the MAETT entries satisfy:*

$$\textit{entry-stg}(E_1) \neq \textit{entry-stg}(E_2)$$

then the stages of updated entries E_1' and E_2' must satisfy:

$$\textit{entry-stg}(E_1') \neq \textit{entry-stg}(E_2')$$

unless E_1' and E_2' represent retired instructions.

The sublemma 5 is proved by case analysis depending on the initial stages of E_1 and E_2. If we tried to proved this lemma with the ACL2 theorem prover, most of the cases are resolved automatically. However, some cases require further analysis of the micro-architectural design. For instance, when we have instructions at pipeline latches at LD1/LD2 and ML2/ML3 simultaneously, both instruction may advance to pipeline latch EX/WB, causing a structural hazard. Actually, this cannot happen because the issuing logic does not allow any instruction issue leading to such a situation. The architectural design assures that only ST instruction can be in LD1/LD2 when ML2/ML3 is also occupied. Since ST instruction does not advances to pipeline latch EX/WB, it will not cause a structural hazard. This fact have to be proven by another lemma before proving Sublemma 5:

Lemma 6 *Suppose E_1 and E_2 are MAETT entries corresponding to a micro-state s, and satisfy values-ok-for?(E_1,s) and values-ok-for?(E_2,s). If the stage of E_1, entry-stg(E_1), is LD1/LD2, and the stage of E_2, entry-stg(E_2) is ML2/ML3, then opcode(entry-inst(E_1)) = 2, indicating E_1 represents an ST instruction.*

```
(defthm LD1/LD2-ML2/ML3-exclusive
    (implies (and (micro-state-p s)
  (entry-p e1)
  (entry-p e2)
  (values-ok-for? e1 s)
  (values-ok-for? e2 s)
  (equal (entry-stg e1) *LD1/LD2*)
  (equal (entry-stg e2) *ML2/ML3*))
     (equal (opcode (entry-inst e1))
    2))

(defthm distinct-stages-preserved
    (implies (and (micro-state-p s)
  (entry-p e1)
  (entry-p e2)
```

```
  (values-ok-for? e1 s)
  (values-ok-for? e2 s)
  (not (equal (entry-stg e1) (entry-stg e2)))
  (not (equal (entry-stg (step-entry e1 s)) *retire*)))
     (not (equal (entry-stg (step-entry e1 s))
 (entry-stg (step-entry e2 s)))))
```

Another lemma we have to prove for Lemma 2 is:

Lemma 7 *Suppose s is a micro-state and M is the corresponding MAETT representation satisfying MAETT-ok?(M,s). Then the updated MAETT MAETT-step(M,s) satisfies no-WAW-hazard?(MAETT-step(M,s)).*

This lemma implies that the next micro-state micro-state-step(s) suffer no WAW-hazard.

Since `no-WAW-hazard?` is a recursive predicate checking whether any MAETT entry pair satisfies `WAW-violation?`, Lemma 7 can be proven easily from the following lemma:

Lemma 8 *Suppose s is a micro-state and M is its corresponding MAETT representation. Further assume E_1 and E_2 are member entries of M satisfying:*

$$\textit{in-order-issue-pair?}(E_2,\ E_1)$$

$$\neg\,\textit{WAW-violation?}(E_1,E_2).$$

Then the updated MAETT entry $E_1' =$ step-entry(E_1,s) and $E_2' =$ step-entry(E_2,s), which correspond to the next micro-state step-entry(s), satisfy:

$$\neg\,\textit{WAW-violation?}(E_1,E_2)$$

2.9 Equivalence Proof of ISA and Micro-architecture

We have proved the equivalence of our ISA and our micro-architectural design in two ways; one approach proves Diagram(a) in Figure ??, and then we verify Diagram(b) from (a) by induction. Our second approach is a direct proof of the correctness criteria given in Diagram(b) without proving Diagram(a). Here we discuss only the latter approach, because it turned out be simpler than the former approach as it does not involve pipeline flushing.

We pictorially portray the proof of the correctness criteria using Figure 4. Let us consider n micro-architectural state transitions $P_0, P_1, \ldots, P_n$. We assume that the initial state P_0 and the final P_n are flushed, as shown in Diagram (b). We further assume that m instructions $I_1, \ldots, I_m$ are executed in the micro-state transitions from P_0 to P_n. Let $S_0 = \texttt{proj}(P_0)$; that is, the ISA state projected from P_0. And let $S_0, S_1, \ldots, S_m$ be the m-step ISA-state transitions from S_0. Now we need to prove $\texttt{proj}(P_n) = S_m$ to satisfy Diagram(b).

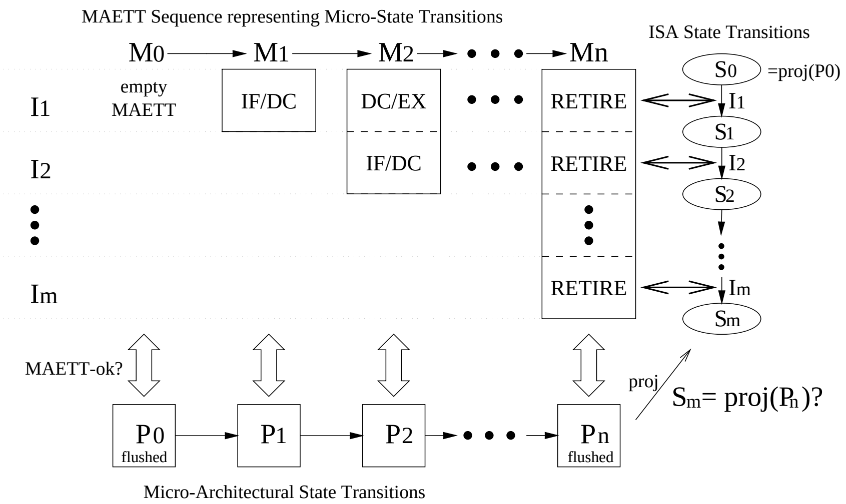

Figure 4: Pictorial Description of the Proof of Diagram(b)

From Lemma 2 and Lemma 3, states $P_0, P_1, \ldots, P_n$ have corresponding MAETT representations $M_0, M_1, \ldots, M_n$ and `MAETT-ok?`(M_i, P_i) is satisfied by each pair of P_i and M_i. Figure 4 shows MAETT's $M_0, \ldots, M_n$, which contain entries representing instructions $I_1, \ldots, I_m$. In the figure, the current stages of each instruction are shown; for instance, I_1 advances from stage IF/DC to DC/EX, and RETIRE's by the time it reaches M_n. M_n contains entries representing all $I_1, \ldots, I_m$ and they are all retired since the corresponding micro-state P_n is flushed. The figure also shows the relations of ISA-state transitions $S_0, \ldots, S_m$ and instructions $I_1, \ldots, I_m$; the ISA-machine executes instruction I_i in the transition from S_{i-1} to S_i. In other words, S_i is the ideal ISA state after executing instructions through I_i. We define a function `final-state(`P_i`,`M_i`)` which returns the ideal ISA state after executing through the last instruction in M_i. So `final-state(`P_n`,`M_n`)` $= S_m$, because I_m is the last instruction in M_n. The definition of `final-state` is easy as a MAETT contains the ideal ISA states corresponding to the instructions. We then can prove:

Lemma 9 *If a micro state P_i and its MAETT M_i satisfies* `MAETT-ok?(`P_i`,`M_i`)`*, and all the instructions in M_i are retired, then* `final-state(`P_i`,`M_i`)` $=$ `proj(`P_i`)` *holds.*

Lemma 9 can be fairly easily proven from the properties `correct-pc?`, `correct-regs?` and `correct-mem?`, which are conjuncts of `MAETT-ok?`. By applying the lemma to M_n and P_n, we will get:

$$S_m = \texttt{final-state}(P_n, M_n) = \texttt{proj}(P_n).$$

File Name	Lines	Time(sec)	Comment
b-ops-aux-def.lisp	39	1.68	Supplementary definitions of bit operation.
b-ops-aux.lisp	298	22.08	Basic Lemmas about Bit Operations.
basic-def.lisp	224	40.74	Definition of Register File and Memory.
diagrams.lisp	1292	215.68	Proof of Pipeline Correctness Diagram.
drain-step.lisp	139	4.45	Definition of Stalling Cycle.
invariants.lisp	27	0.41	Proof of Lemma 1 from Sub-lemmas.
invariants1.lisp	5171	1742.39	Proof of Properties in Appendix A.
invariants2.lisp	1222	340.78	Proof of Properties in Appendix A.
invariants3.lisp	1256	228.74	Proof of Properties in Appendix A.
p-drain.lisp	489	48.75	Definition of Pipeline Flushing.
pipeline-def.lisp	1351	109.97	Definition of Pipeline Design.
reg-modifier.lisp	299	132.65	Theory of Register Modifying Instructions.
seq-def.lisp	157	10.57	Definition of Sequential Machine.
table-def.lisp	1628	103.51	Definition of Instruction Table.
table-lemmas.lisp	3871	549.72	Basic Lemmas about Instruction Table.
table-misc.lisp	192	29.12	Miscellaneous Lemmas including Lemma 2.

Table 2: Lines and Proof Time for the Script Files. The CPU time for definition and proof is measured on a Pentium 133MHz, Solaris 2.5 Machine. It does not include time for loading ACL2 certified books.

3 Mechanical Check of the Proof

Our pipeline correctness proof has been mechanically verified by ACL2 theorem prover. The number of the lines of the proof scripts and the CPU time for the proof are given in Table 2. It took less than 1 man-month to design the machine and debug it, then took 3 man-months to verify it. Since we have tried many experimental techniques in our project, and the machine is very small compared with practical commercial processors, it is too early to make a conclusion on the labor cost. In our future work, we will pay close attention to the labor involved.

There are a few other points we want to discuss here.

The execution capability in ACL2 allows us to run the behavioral description of our pipelined machine. Using this feature, we have run small programs on our machine design and discovered most of our design errors before applying formal verification techniques. Actually, all the fatal error except one were eliminated in this stage. The only fatal error uncovered by the formal verification concerned undefined instructions, which we forgot to test in the debugging phase. We believe it is important to remove most of the design errors using techniques like simulation before attempting to apply formal verification techniques.

The actual verification work did not proceed quite as directly as we described in the last few sections. We first proved the pipeline correctness diagram under the assumption that Lemma 1 holds, then we proceeded to the proof of

Lemma 1, which took most of our verification effort. During this proof, we often discovered that we needed to prove additional properties about the pipeline and instructions, or that some naively defined properties do not hold with our machine. In order to reflect these findings, we continuously modified the definition of the table-updating function and properties in Appendix ??. The predicate table-of?(), which is the conjunction of all the resulting properties, is strong enough to prove the pipeline correctness diagram, but weak enough to hold for any reachable pipeline states. This definition is a result of our interaction with the theorem prover.

Our method checks whether the micro-architectural components work cooperatively. One of the most interesting minor defects we found involved a bookkeeping register for the issuing logic. The issuing logic reads and updates a register to record the pipeline configuration, but the writeback unit also refers to this register to determine the source of the next written-back datum. To our surprise, the designs of the issuing logic and the writeback unit used different assumptions about the bookkeeping register. Although the problem did not turn out to be an major error, it demonstrated that such conflicting assumptions in the design of micro-architectural components can be uncovered by our approach.

4 Conclusion

We have verified the behavioral equivalence of our pipelined machine with respect to its sequential specification using ACL2 theorem prover. Our machine design is certainly not the most complex pipelined machine that has been formally specified and verified. However, our machine has more complex pipeline control and interlocks than those in earlier verification research we are aware of. Our example machine features out-of-order completion of instructions, speculative instruction fetching and dynamic resolution of RAW-hazards, WAW-hazards and structural hazards.

For the sake of the verification, we introduced the instruction table as a new level of machine abstraction. It allows us to directly reason about the instructions executed by the pipeline machine. With help of the table representation, we defined various properties about the pipelined machine design. These properties are proved to hold for reachable pipelined states. The proof are conducted incrementally by examining the related micro-architectural components.

We adopted a correctness criteria that is defined using the idea of pipeline flushing. This allowed us to express the correctness of the pipelined machine containing complex control logic. The correctness criteria is verified using the pipeline properties of the reachable pipeline states. Since this approach does not appear to suffer the computational cost explosion of the other methods, we hope that our methodology will allow us to verify bigger and more complex examples.

References

[1] M. Agaard, M. Leeser, Reasoning About Pipelines with Structural Hazards, Pipelined Microprocessors, Theorem Provers in Circuit Design : theory, practice, and experience, Lecture Notes in Computer Science 901, Springer Verlag, 1995, page 13-32.

[2] R. S. Boyer and J S. Moore. A Computational Logic Handbook. Academic Press, Boston, 1988.

[3] J. R. Burch, D. L. Dill: Automatic Verification of Pipelined Microprocessor Control, In D. Dill Editor, Computer Aided Verification, Lecture Notes in Computer Science 818, Springer Verlag, 1994, page 68-80.

[4] J. R. Burch. Techniques for verifying superscalar microprocessors. In Design Automation Conference, June 1996.

[5] D. Cyrluk. Microprocessor verification in PVS: A methodology and simple example, Technical Report SRI-CSL-93-12, SRI Computer Science Laboratory, Dec. 1993

[6] J. Hennessey, D. Patterson, Computer Architecture a Quantitative Approach, Morgan Kaufmann Publishers, Inc., 1996.

[7] W. A. Hunt, Jr., B. 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, page 35-48. Prentice-Hall International Series in Computer Science, Engle wood Cliffs, N.J., 1992

[8] R. B. Jones, D. L. Dill, J. R. Burch, Efficient Validity Checking for Processor Verification, 1995 IEEE/ACM International Conference on Computer-Aided Design, pages 2-6.

[9] M. Kaufmann, J S. Moore, ACL2: An Industrial Strength Version of Nqthm, Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS-96), pages 23-34 , IEEE Computer Society Press, June 1996.

[10] M. Srivas, M. Bickford, Formal Verification of a Pipelined Microprocessor, IEEE Software, September 1990, page 52-64.

[11] M. K. Srivas, S. P. Miller, Formal Verification of a Commercial Microprocessor, Technical Report SRI-CSL-95-12, SRI Computer Science Laboratory, July 1995.

[12] S. Tahar, R. Kumar, Formal Verification of Pipeline Conflicts in RISC Processors, Proc. European Design Automation Conference (EURO-DAC94), Grenoble, France, September 1994, IEEE Computer Society Press. page 285-289.

[13] P. J. Windley, M. L. Coe, A Correctness Model for Pipelined Microprocessors, Theorem Provers in Circuit Design : theory, practice, and experience, Lecture Notes in Computer Science 901, Springer Verlag, 1995, page 33-51.

[14] P. J. Windley, J. R. Burch: Mechanically Checking a Lemma Used in an Automatic Verification Tool, Formal Methods in Computer-Aided Design, Lecture Notes in Computer Science 1166, Springer Verlag, 1996, page 362-376.