Results from Verifying a Pipelined Microprocessor

A Thesis

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

by

Michael L. Coe

October 9, 1994

Major Professor: Phillip J. Windley, Ph.D.

Authorization to Submit
Thesis

This thesis of Michael L. Coe, submitted for the degree of Master of Science with a major in Computer Science and titled "Positive Results from Verifying a Pipelined Microprocessor" has been reviewed in final form, as indicated by the signatures and dates given below. Permission is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor ______________________ Date ____________
Phillip J. Windley

Committee Members ______________________ Date ____________
Michael Barnett

______________________ Date ____________
James Frenzel

Department Administrator ______________________ Date ____________
John Dickinson

Discipline's College Dean ______________________ Date ____________
Richard Jacobson

Final Approval and Acceptance by the College of Graduate Studies:

______________________ Date ____________
Jean'ne M. Shreeve

Abstract

Hardware verification has become a very active area of research in the last few years. As computers and other electronic devices push their way into mainstream society, the need for high quality, ultra reliable hardware has become greater. Computers now control safety critical applications like the flight control systems of jumbo jets. Safety critical applications such as these highlight the importance of the quality of the computer systems that control these systems. Our research here is intended to devise new technologies and techniques for developing more reliable computing systems. In this thesis we present research in the area of hardware verification–specifically microprocessor verification.

The microprocessor is the heart of the modern day computer. If we wish to build reliable computers we must start with the microprocessor. There has been much research in this area already. However, most of the research examples and case studies were done to develop methods and strategies for verifying microprocessors. Because of this, most of the examples of verified microprocessors are very simple even compared to the processor that runs most of the personal computers used today. Only recently have methodologies been developed to verify microprocessors of the complexity that are found in industry. This thesis presents the results of applying state of the art verification techniques to a very modern microprocessor.

In this thesis we present the formal verification of a pipelined microprocessor named SAWTOOTH. SAWTOOTH is the successor to Windley's verified microprocessor AVM-1[Win90b]. SAWTOOTH is very much like modern microprocessors. The processor contains advanced features such as a five stage pipeline and data forwarding hardware

The formal specification and verification of the processor is done in the **HOL** theorem proving system. Windley's generic interpreter theory (GIT) is used for the specification, however it was not used to complete the verification due to the complexities of the data and temporal abstraction that take place when five instructions execute simultaneously.

The EBM specification for SAWTOOTH was written from a hand translation of the designers VHDL while the designer was still designing the processor. This gave the specifier a much greater understanding of the microprocessor, and it also helped uncover design errors prior to fabrication.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Computers are found in almost every facet of modern society. Microprocessors and the software they execute can be found today in devices ranging from Nintendo games to the space shuttle. The application of computers on the space shuttle and systems like it bring up a very important concern about the growing use of computers: when people's lives are on the line, we want to know that the software and hardware controlling these computers work correctly and will work correctly every time. A hardware or software failure in the systems on the shuttle could be catastrophic. There has been much work done in the area of verifying the correctness of software through vigorous testing and verification of code. Hardware verification is an area of research that has just come into the spotlight in the last few years. Even if you have verified your software, line by line, running this software on a microprocessor with faults in it could still lead to undesirable results. For this reason, we focus our research on the verification of microprocessor systems.

Hardware and microprocessor verification has become an active area for research in the last decade. There have been several examples of verified microprocessors in the literature [Coh88b, Hun87, Hun89, Joy89, BG90, SB90a, Win90b]. However most of these verified microprocessors were very simple processors chosen to illustrate new research. The most complex processor cited above is AVM-1 which was verified by Windley [Win90b] to demonstrate the generic interpreter theory [Win90a]. The generic interpreter theory is a methodology which structures the verification effort.

In this thesis we present the formal specification and verification of a pipelined microprocessor similar in complexity to a MIPS R2000. The processor was designed by a graduate student working in electrical engineering. One unique facet of this verification effort is that the specifications were developed *while* the designer was completing the design. The verification had a positive impact on the design by uncovering some design errors very early in the design process.

The specification of SAWTOOTH is created using the generic interpreter theory [Win90a]. As with previous microprocessor verification efforts, the specification is broken down into a series of increasing abstract levels. For SAWTOOTH we define an electronic block model, a phase level description, a pipeline level description, and a architectural level specification. The verification of the lower levels also used the generic interpreter theory, however for the crucial step of verifying that the pipeline level implements the macro level instructions the generic interpreter theory is not used. The generic interpreter theory fails because of the temporal and data abstraction involved with the concurrent execution of instructions.

1.2 Hardware Verification

With the advent of very fast computer systems came the ability to develop systems to better analyze hardware designs. The traditional approach to validate hardware designs is through the use of simulators. But as the complexity of hardware and microprocessor designs increase, the tractability of fully simulating a design decreases. Thus formal hardware verification came into the being as an alternative method to insure the correctness of hardware designs.

Camurati and Prinetto [CP88] give two definitions of the functional correctness of hardware. The definition we present here summarizes clearly the process of hardware verification.

> What the system should do is called its *specification*, while any one of the possible devices that realizes the specification is called an *implementation*.... Any piece of hardware is functionally correct if we can somehow prove that its implementation realizes the specification.[CP88]

It is important to note here that verification only validates the specification–not the actual hardware. Therefore the critical issue in formal methods and verification is the correct modeling of the microprocessor. Warren Hunt describes the importance of his verification of FM8501 as follows:

> The primary contribution of the FM8501 effort is to show that a microprocessor or like device can be better described with several pages for formulae instead of hundreds of pages of text. The compact formal description is better suited to the task of description: it takes less time to read, it is unambiguous, and it provides an accurate basis for building systems that use it.[Hun87]

Hunt strongly asserts that the important achievement of the work was not the verification, but the specification itself. He also stated that formal specifications are better for describing hardware systems. In this thesis we present a side-by-side comparison for a VHDL description and a formal specification of SAWTOOTH. In most cases, the **HOL** specifications are more succinct than the VHDL.

Hardware specifications and verifications are usually done in a theorem proving system. The specifications are written in a formal logic such as predicate calculus or higher order logic. For the work presented here, we used the **HOL** theorem prover. **HOL** was developed at Cambridge University by Mike Gordon[Gor88]. **HOL** stands for higher order logic[Chu40] which is the logic system that is embedded into the theorem prover.

1.3 Conventions

Throughout this document numerous **HOL** definitions, theorems and example sessions are presented. VHDL for the microprocessor is also presented in this thesis. We present this material using the following conventions:

- All **HOL** definitions, theorems, and example sessions will be inside of boxes.
- **HOL** definitions will be preceded by this symbol: $\vdash_{def}$. All **HOL** theorems will be preceded by simply: $\vdash$.
- The VHDL for the SAWTOOTH microprocessor will also presented in boxes, but in the `verbatim` font.
- All **HOL** example sessions will also be presented in the `typewriter` font.
- All **HOL** definitions or theorems that are referenced in the text will be done in the `typewriter` font.
- All instruction abbreviations will be listed in all capital letters. For example, the jump instruction is abbreviated with JMP.
- The complete specification for SAWTOOTH is broken into a series of increasing abstract levels. Throughout the specification several **HOL** definitions are used to aid in the readability of the specifications. Definitions that are not found in the main text of this thesis can be found in appendix C.

1.4 Contributions

This thesis presents some contributions to the field of microprocessor verification:

- We believe, to the best of our knowledge, that SAWTOOTH is the most complex microprocessor design to be specified and verified. To date, there have been three pipelined microprocessors verified: Mini Cayagua [SB90a], Uinta [WC94], and Tahar and Ramayya's DLX clone [HP90]. Mini Cayuga was a simple pipeline design with only a three-stage pipeline. Uinta has a five stage pipeline and 2 levels of forwarding hardware, but it lacks support for external interrupts. Tahar and Ramayya have also done some verification work with a DLX [HP90] like processor, however they have only verified the RISC core of the processor.

- The research here presents new verification strategies that were originally developed by Windley[WC94], but were extended for a slightly different, and more complex pipeline design. The major difference between the Uinta proof and this verification is how each processor deals with control hazards caused by branch instructions. The details of these new verification strategies are shown in section 6.3.

- This specification and verification effort was done *while* the designer was completing the hardware design. The design and verification were done by independent parties, but there was interaction between the two parties. Our research shows that carrying out a design specification and verification concurrently to design can be beneficial. We discovered several design errors in the SAWTOOTH microprocessor that were missed under standard simulation. These errors which lead to software constraints are listed in section 6.1.

1.5 Contents

The rest of this thesis will be presented as follows. Chapter two will describe previous research in the area of microprocessor verification. Chapter three will describe the specification of the electronic block model of SAWTOOTH. Chapter four will present the specification and verification of both the phase and pipeline level specifications for SAWTOOTH. Chapter five will present the specification of the assembly language level of SAWTOOTH. And finally in Chapter six the results of our verification efforts will be presented.

Chapter 2

Previous Work

In this section we present a summary of the research that has been done in the area of microprocessor verification. Microprocessor verification is merely a special kind of hardware verification. The hardware verification presented here is done in the **HOL** theorem proving system which was developed by Mike Gordon [Gor88]. Pioneering work in the area of hardware verification in **HOL** was done by Camelleri *et al.* [CGM87], and Melham investigated the different types of abstraction necessary in hardware verification[Mel88].

Other verification systems have been used to research hardware and microprocessor verification. These systems include: The Boyer-Moore theorem prover [BM79], the VERIFY system [Bar84], the Clio verification system [BMS89], and the Nuprl theorem prover [Lee92]. A full description of these systems and the research done in these systems is beyond the scope of this thesis. However a brief description of the **HOL** system is in order to aid in reading the rest of this thesis.

2.1 HOL

Windley [WC94] presents a brief and concise description of the **HOL** theorem proving system. This section is taken from that reference.

To ensure the accuracy of our specifications and proofs, we developed them using a mechanical verification system. The mechanical system performs syntax and type checking of the specifications and prevents the proofs from containing logical mistakes. The HOL system was selected for this project because is has higher-order logic, generic specifications and polymorphic type constructs. These features directly affect the expressibility of the specification language. Furthermore HOL is widely available, robust, and has a growing world-wide user base. However, there is nothing our work that requires the HOL theorem proving system.

HOL is a general theorem proving system developed at the University of Cambridge [CGM87, Gor88] that is based on Church's theory of simple types, or higher-order logic [Chu40]. Similar to predicate logic in allowing quantification over variables, higher-order logic also allows quantification over predicates and functions thus permitting more general systems to be described.

For the most part, the notation of HOL is that of standard logic: $\forall$, $\exists$, $\wedge$, $\vee$, etc. have their usual meanings. There are a few constructs that deserve special attention due to their use in the remainder of the paper:

- HOL types are identified by a prefixed colon. Built–in types include `:bool` and `:num`. Function types are constructed using $\longrightarrow$. HOL is polymorphic; type variables are indicated by a type names beginning with an asterisk.
- The HOL conditional statement, written `a` $\rightarrow$ `b` | `c`, means "if `a`, then `b`, else `c`."
- The construct `let v1 = expr1 and v2 = expr2 and ... in` simultaneously defines local variables `v1`, `v2`, etc. with values `expr1`, `expr2`, etc.
- Comments in HOL are enclosed in percent signs, `%`

2.2 VHDL

VHDL is the hardware description language used to design SAWTOOTH. Throughout chapter three, we compare the designer's VHDL code with our **HOL** specifications, therefore a brief summary of VHDL is necessary. While a full description of VHDL is certainly beyond the scope of this thesis, we present a few basic ideas which will aid in reading. Shown below is a short example of behavioral VHDL code. the VHDL presented in this thesis for SAWTOOTH is behavioral VHDL and it will look much like this example (although a bit more advanced).

VHDL stands for VHSIC Hardware Description Language. VHDL was developed by the Very High Speed Integrated Circuits (VHSIC) Program Office of the Department of Defense. Work started on VHDL in 1981 and after several revisions of the language, the IEEE standardized the language in the standard IEEE-1076[Soc88].

VHDL is a language created for designing digital devices. One of the advantages of VHDL is that it is independent of any single design methodology[LSU89]. To do this, the basis of the language is an abstraction of how digital devices work. The abstraction includes three models of digital devices: behavioral, timing, and structural.

The basic elements of a hardware device in VHDL are the *component*, the *port*, and the *signal.* The component is the building block of the device. A component can range from a gate to an entire board. The *port* is the place where data flow in and out of a component. A signal is the path from one component to another. Components are connected by signals at their ports.

A component is defined by a *design entity.* A design entity is made up of two parts: an entity declaration and a architecture body. The entity declaration provides a black box view of the component.

A entity declaration declares a component's in and out ports. In the example shown below[LSU89], the entity specification states that there are four ports. Two of these ports are for input: X and Y. The half-adder also has two output ports: the sum and carry results. The entity specification also states that the information flowing through all of these ports is one bit wide. The architectural body describes the internal view of the component. It describes the behavior of the device. For example, the architectural specification below describes the internal actions inside a half-adder:

```
-- The entity declaration
entity Half_adder is
  port (
    X : in Bit ;
    Y : in Bit ;
    Sum : out Bit ;
    Carry : out Bit) ;
end Half_adder ;


--The architecture body:

architecture  Behavioral_description of Half_Adder
begin
  process
   Sum  <=  X  xor Y after 5 Ns;
   Carry <= X  and Y after 5 Ns;
  wait on X,Y ;
  end process;
end  Behavioral_description
```

This example, given by Lipset *et al.*[LSU89], says that the signal connected to the out port Sum is assigned the "exclusive or" of the value of the signal connected to the in port X and the signal connected to port Y, after a delay of five nanoseconds. The first line in the example is a comment line. On a given line of text, all the characters following a "–", are comments.

A full description of VHDL is not necessary to read this thesis, but interested readers are urged to read some of the many texts on VHDL[LSU89, Coe89].

2.3 Data and Temporal Abstraction

In our verification efforts two types of abstraction are used frequently: data and temporal abstraction. For both types of abstraction they are used in a context of relating two levels of specification where one is more abstract then the other.

Data abstraction relates what part of the microprocessor state is visible between specification levels. For example, for a microprocessor, the state of the processor at the assembly language level(or macro level) usually contains a program counter, a register file, memory, and a program status word. However each assembly language instruction could be implemented by a series of microinstructions. At the microprogram level, the state contains such items as the microprogram counter, the store memory address register, and other pieces state not visible to the assembly language programmer. When we specify the macro level we have, through data abstraction, hidden the implementation details of each instruction.

Temporal abstraction relates the granularity of time between two specifications. Continuing with the example started above, the assembly language programmer considers one unit of time to be how long it takes to execute one instruction. However at the microprogram level, each microinstruction takes one clock cycle to execute, and each assembly language instruction is implemented by a series of microinstructions. So between the microprogram level and the macro level we have temporal abstraction.

2.4 Verified Microprocessors

2.4.1 FM8501 and FM8502

Warren Hunt verified a microprocessor called FM8501 [Hun87] for his Ph.D. dissertation. FM8501 is a 16-bit microprocessor. All instructions for the processor are 16 bits in length, the processor uses word addressing only where the size of each word is 16 bits, and the processor has eight general purpose 16-bit registers. Also at the programmer's view of the processor are four status flags: carry, overflow, negative, and zero. The FM8501 is a very CISC-like processor. It supports four different addressing modes, and instructions fall into one of three categories: register-register, register-memory, or memory-memory.

The specification and verification of FM8501 was done in the Boyer-Moore theorem prover [BM79]. For the specification, two views of the microprocessor were described. The first view is that of the hardware. The sequential hardware logic was modeled with a set of recursive functions written in the logic of the theorem prover. These functions were put together to create one state machine. The second view of the processor was that of the programmer's model. For this specification Hunt defined an instruction interpreter. To verify the processor, Hunt proved that if you start with a given state and execute a list of n instructions, then the resulting state will be equal to that of the state resulting from executing the hardware interpreter.

Hunt continued work in this area by verifying the FM8502 microprocessor[Hun89]. FM8502 is an extension of the processor design for FM8501. All registers and instructions for FM8502 are 32-bits. Besides the fact that the registers are 32-bit registers, much of the design for FM8502 is similar to that of FM8501. The verification of FM8502 was carried out in a similar manner to that of FM8501.

2.4.2 Viper

Viper is a microprocessor that was designed by the Royal Signals and Radar Establishment. The intent of the project was to build a verified microprocessor for use in safety critical systems such as nuclear reactors and avionics in both civil and military aircraft.

Viper is the first commercial microprocessor to be designed and fabricated using formal methods. To aid in this goal, the design of Viper was kept simple. Viper [Cul88] is a 32-bit microprocessor with a 20-bit address space. The processor has a total four registers: an accumulator, A, a program counter, P, and two index registers, X and Y. The processor has no support for a stack or interrupts.

The design for the microprocessor was developed in a series of decreasingly abstract levels, ranging from a top level functional description to a circuit level description. The specifications for the upper levels were originally written in the logic of Robin Milner's LCF_LCM theorem prover. Avra Cohn [Coh88b] translated these specifications into **HOL** and verified the upper two levels. Cohn [Coh88a] however could not complete her verification because the proof that the electronic block model realized the state machine specifications became too complex to finish with the tools and methodologies available at the time.

The incomplete verification effort of Viper spurred interest in the project at the University of California, Davis. The Systems Verification Laboratory at Davis noted that one of the major causes of the failure of the viper proof was that the jump in abstraction between the electronic block model and the programmer's model was too great. Using this fact, they designed a simpler electronic block model for Viper that realized the original instruction set. Using this new electronic block model, Arora *et al.*[ALL+93] verified that their new EBM implemented the original RSRE instruction set for Viper using the generic interpreter theory[Win90a, Win93](the generic interpreter theory will be described in more detail in section 2.5.1).

2.4.3 Tamarack

Tamarack is a very simple microprocessor with only eight instructions. It is microprogrammed machine with less then 15 internal registers, an ALU unit, hardware to allow interrupts, and one system bus shared among all of these. Because of the simplicity of its design Tamarack has become an excellent example for hardware verification.

An early version of Tamarack was first verified by Michael Gordon [Gor83] using the LCF_LCM theorem prover. This version was also verified by H. Barrow using the VERIFY system [Bar84]. Jeff Joyce translated Gordon's specifications for Tamarack in **HOL** [Joy88], and he also had the microprocessor fabricated into CMOS. For his Ph.D. thesis, Joyce [Joy89] verified Tamarack from the electronic block model to the macro programming level. Coe and Windley [CW92] verified Tamarack once again to demonstrate the generic interpreter theory.

2.4.4 SECD

SECD is a microprocessor designed and verified at the University of Calgary by G. Birtwistle and B. Graham [BG90]. The processor is an implementation of Henderson's [Hen80] variant to the abstract

Lisp machine created by Landin [Lan64] to reduce lambda expressions.

The SECD machine has four main registers from which it derives it's name: **S**, **E**, **C**, and **D**. Each register is actually holds a memory address and implements a stack. For this project, Graham *et. al.* chose to implement the processor as a coprocessor to a larger system. This reduces the number of tasks the processor had to perform and simplified the verification. Designing for ease of implementation and verification was a key idea for this project. Following this theme, the processor uses a a single bus architecture and an external RAM memory with each word being 32 bits wide.

The specification and verification were done in a hierarchical fashion as was done by Cohn [Coh88b] and Joyce [Joy88]. The specification was broken down into three levels: a Low-Level Definition, a Register Transfer Level, and the Top Level Specification. To date, Graham and Birtwistle [BG90] have verified that the Register Transfer Level correctly implements the Top Level specifications. The SECD processor has been fabricated.

2.4.5 The Sobel Image Processing Chip

The Sobel Image Processing Chip was designed at Research Triangle Institute. The processor is an implementation of the Sobel edge detection algorithm. It contains roughly 10,000 transistors, and is divided into two components. The first component is a *window processor* which filters images and generates four values representing the differences in intensity along the vertical, horizontal, right diagonal, and left diagonal. These values are then passed to the *magnitude-and-direction processor* which calculates gradient magnitude and gradient direction.

Narendran and Stillman verified this processor [NS89] using a methodology different from the others listed so far. They use an equational method for theorem proving that was developed by Narendran and Kapur [KN85]. The verification naturally broke down into verifying the correct behavior of the two sub-processors.

2.4.6 AVM-1

To complete his Ph.D. thesis, Windley [Win90a] verified a microprocessor called AVM-1 (A verified microprocessor). AVM-1 has a RISC-like architecture, however it was designed to be a microcoded processor for ease of verification. AVM-1 has 24 general purpose registers and seven supervisory mode registers–all of which are 32 bits wide. The instruction set contains 30 instructions with room to expand to 64. The processor handles interrupts and uses a synchronous memory model.

The specification and verification of AVM-1 was done using the generic interpreter theory [Win90a, Win93] which is described in section 2.5.1. The specification for AVM-1 was broken down into four levels: an electronic block model, a phase level interpreter, a microprogram level interpreter, and a macro-level interpreter. The macro-level specification is the assembly language programmer's view of the microprocessor.

2.4.7 Mini Cayuga

Mini Cayuga is a scaled down version of the RISC processor Cayuga which was designed in a graduate level VLSI course at Cornell University. [SD88] Mini Cayuga contains 32 general purpose registers all 32 bits wide ; the processor also utilizes several special purpose registers including a *Nxpc* (next program counter) register. The instruction set of Mini Cayuga is a subset of the original RISC processor, however each class of instructions for Cayuga is represented in Mini Cayuga. The microprocessor assumes that the cache (if any) is not on the processor, and the memory interfaces synchronously with the CPU. Mini Cayuga does retain support for one type of interrupt.

Mini Cayuga differs from the other microprocessors mentioned thus far because it is a pipelined microprocessor; The processor has a three stage instruction pipeline. The three stages of the pipeline are *instruction-fetch*, *compute*, and *write-back*. There are four types of instructions for Mini Cayuga: arithmetic, jump, load, and store. The arithmetic and jump instructions take three cycles to execute, so in an ideal situation one arithmetic/jump instruction could be completed during every clock cycle due to the three stage pipe. However, load and store operations take four cycles due the necessary reference to memory. Interrupts can occur at any time during the execution of instructions. When an interrupt is signalled, the processor finishes executing the instruction farthest into the pipe, flushes the pipe, and starts executing a service routine. Currently Mini Cayuga supports only one interrupt–reset.

Srivas and Bickford [SB90a] verified Mini Cayuga using the Clio verification system [BMS89]. Although the verification tool was quite different from the others listed above, the verification was undertaken in much the same manner as that of Tamarack and Viper. Two descriptions of Mini Cayuga were used: an *abstract* specification describing the programmer's view of the processor, and a *realization* specification of the circuit design. To verify the processor, Srivas and Bickford showed that if both of these specifications were to execute the same program, after each instruction completed the state of the realization would equal the state of the abstract model (taking into account data abstraction). An important point to note about the proof is that between the two models, Srivas and Bickford did not use any temporal abstraction [SB90b] as done in previous microprocessor verifications [Coh88b, Joy88, Win90a]. In these previous efforts the length of a discrete unit of time at the top level was the length of time to execute an instruction. For Mini Cayuga, Srivas and Bickford retained the time used for the *realization* specification because they felt that it was necessary to specify the correct behavior of interrupts.

This verification is important because it is the first pipelined microprocessor ever verified. However, the processor design is still simple compared to commercial microprocessors. The processor uses only one level of forwarding, and has no hardware support for early branch detection.

2.4.8 Uinta

Uinta is a pipelined microprocessor verified by Windley and Coe [WC94]. Uinta contains many architectural features found in many modern processors such as a five stage instruction pipeline and two levels of data forwarding hardware.

The specification of Uinta was broken down into four levels: the electronic block model, the phase level specification, the pipeline level specification, and the architectural level (assembly language) specification. The proofs that the EBM implements the phase level and the phase level implements the pipeline level are done using standard generic interpreter theory methods (see section 2.5.1). The important contribution of this work was the correctness model which allows the final results to be used in the same way that past correctness results have been used without changing other methodologies. For the top level verification, the generic interpreter theory was not used (for a complete description of why this is so see section 6.3). Instead, Windley proved that the pipeline specification implemented each of the assembly language instructions separately. Windley used a temporal abstraction function similar to one used by Tahar and Kumar used in their new RISC model described in the next section. However, what differentiates the Uinta proof from the Tahar and Kumar verification is that the Uinta proof requires no assumptions about the correct pipelining of instructions to eliminate hazards.

We will not cover the details of this verification here since many of the techniques used in the Uinta verification are also used to verify SAWTOOTH. These techniques will be discussed further in section 6.3.

2.5 Methodologies for Microprocessor Verification

2.5.1 The Generic Interpreter Theory

Even though the verification efforts listed above differ greatly, some even use different verification tools, their specifications are somewhat similar. They each break down the specification into a hierarchy of decreasingly abstract levels; each level of this hierarchy is usually modeled as a state transition system; and verification is usually undertaken by showing that a lower level specification implements (implies) a higher more abstract level. Windley [Win90a] noticed these similarities and generalized the whole process with the generic interpreter theory.

The generic interpreter theory is a methodology for specifying and verifying microprocessors. To model the state transition behavior often used in microprocessor specification, Windley chose an interpreter. An interpreter is a computing structure with one control point. One of many possible instructions is chosen at this control point based on the current state and inputs. The state is then updated by the chosen instruction and the cycle begins again.

We now present a more detailed explanation of the generic interpreter theory because we will use this model for parts of the specification and verification of SAWTOOTH.

Table 2.1: The abstract functions and their types for the generic interpreter model.

Operation	*Type*
`instructions`	`":*key -> (*state->*env->*state)"`
`select`	`":*state->*env->*key"`
`output`	`":*key->(*state->*env->*out)'"`
`subout`	`":*out'->*out"`
`substate`	`":*state'->*state"`
`subenv`	`":*env'->*env"`
`Impl`	`":(time'->*state')->(time'->*env')->(time'->*out') ->bool"`
`sync`	`":*state'->*env'->bool"`

The Abstract Representation

The formalization is done in the **HOL** theorem proving system [Gor88] using the abstract theory package [Win92]. An abstract theory is parameterized so that some of the types and constants in the theory are undefined inside the theory except for their syntax and an algebraic specification of their semantics. The abstract representation is the interface to the generic model and crucial to the work reported here since it the abstract representation that structures the specification. We specify the abstract representation of the generic interpreter by defining a list of abstract objects and operations. Table 2.1 [1] shows the operations and their types.

The following abstract types are used in the representation.

- `:*state` represents the state.
- `:*env` represents the environment. The environment provides inputs to the interpreter.
- `:*key` is type containing all of the keys. Keys are used to select instructions. For example, the opcodes form the keys at the macro–level.
- `:*out` represents the output of an interpreter.

Using the functions from the abstract representation, we can define an interpreter. The form of this definition is essentially the same as the form of specific microprocessor specifications given in [Coh88b, Hun87, Joy88].

```
⊢def ∀gi s e p.
   INTERP gi s e p =
   (∀t.
     let k = select gi(s t)(e t)
     in
      ((s(t + 1) = instructions gi k(s t)(e t)) ∧
       (p t = output gi k(s t)(e t))))
```

[1] Table 2.1 taken from [Win93].

The specification of an interpreter is a predicate relating the contents of the state stream at time $t+1$ to the contents of the state stream at time t. The relationship is defined using the functions from the abstract representation.

The generic interpreter theory also contains numerous theorems about the abstract representation. The most important is a correctness proof that shows that `Impl` implements `INTERP` when the theory obligations are satisfied. The theory obligations that have to be proven for an interpreter are `INST_CORRECT` and `OUTPUT_CORRECT`. The definitions for these are given below.

```
INST_CORRECT =
⊢def ∀s' e' p' k gi.
    INST_CORRECT gi s' e' p' k =
    implementation gi s' e' p' ⟹
    (∀t.
      let s t' = substate gi(s' t')
      in
       let e t' = subenv gi(e' t')
       in
        let f t' = sync gi(s' t')(e' t')
        in
         ((select gi(s t)(e t) = k) ∧ f t ⟹
          (∃c.
            Next f(t,t + c) ∧ (instructions gi k(s t)(e t) = s(t + c)))))
```

`INST_CORRECT` states that given a `select` function and a synchronization function (`sync`), there exists a time `c` where the implementation level and the interpreter level are synchronized, and at that time (t+c) the state is equal to the output of the `instructions` function.

```
OUTPUT_CORRECT =
⊢def ∀s' e' p' k gi.
    OUTPUT_CORRECT gi s' e' p' k =
    implementation gi s' e' p' ⟹
    (∀t.
      let s t' = substate gi(s' t')
      in
       let e t' = subenv gi(e' t')
       in
        let p t' = subout gi(p' t')
        in
         let f t' = sync gi(s' t')(e' t')
         in
          ((select gi(s t)(e t) = k) ∧ f t ⟹
           (p t = output gi k(s t)(e t))))
```

`OUTPUT_CORRECT` states that given a `select` function and a synchronization predicate, the output of the implementation level filtered through the function `subout` is equal to the output of the interpreter.

Specification and Verification

Specifying an interpreter is done by creating a concrete definition for each of the functions in Table 1. Then, using these concrete definitions, we instantiate a concrete theorem of the form of INTERP. To verify the interpreter, we have to prove the theory obligations for our instantiated interpreter.

For a microprogrammed processor such as Tamarack there are typically three interpreters we specify: the macro programming level, the microprogramming level, and the phase level. Each interpreter specification defines behavior as implemented by a lower level. For example the micro level interpreter defines the behavior of the microinstructions implemented by the phase level. Using this idea we can build a hierarchy of levels. The lowest level in our hierarchy is the hardware definition or electronic block model which implements the phase level interpreter. Using this hierarchy we can break down the verification to a series of smaller proofs.

- EBM $\Longrightarrow$ Phase Level
- Phase Level $\Longrightarrow$ Micro Level
- Micro Level $\Longrightarrow$ Macro Level

From these smaller proofs we can derive the final correctness statement: EBM $\Longrightarrow$ Macro Level. It may be possible to obtain a final correctness lemma by proving the hardware properly implements the assembly language instructions. However this proof would likely be very tedious, very unstructured, and very time consuming. The generic interpreter theory places a structure on the verification; the proof is naturally broken down into smaller, more tractable proofs.

2.5.2 Tahar and Kumar's RISC Methodology

Sofiéne Tahar and Ramayya Kumar developed a methodology like the generic interpreter theory provides a general model that can be followed to verify pipelined RISC cores [TK93]. The research is built upon Windley's interpreter theory but is specialized for pipelined microprocessors. Using this new methodology, Tahar and Kumar are currently verifying a processor designed after DLX [HP90].

The RISC methodology, like the generic interpreter theory, breaks down the specification of the processor into a hierarchy levels. Figure 2.1 shows the hierarchy that Tahar and Kumar use to specify RISC cores [2]. The Phase level describes what actions occur during one phase of a clock cycle. The Stage Level described what actions take place during each stage of the pipeline. The Class Level is perhaps the most important contribution of the work of Tahar and Kumar. Normally to verify the Instruction level, one instruction definition is made for each processor instruction. However, Tahar and Kumar noticed the fact that in pipelined processors, most instructions in a class (ALU, BRANCH, ALU Immediate)

[2]Figure 2.1 was taken from [TK93].

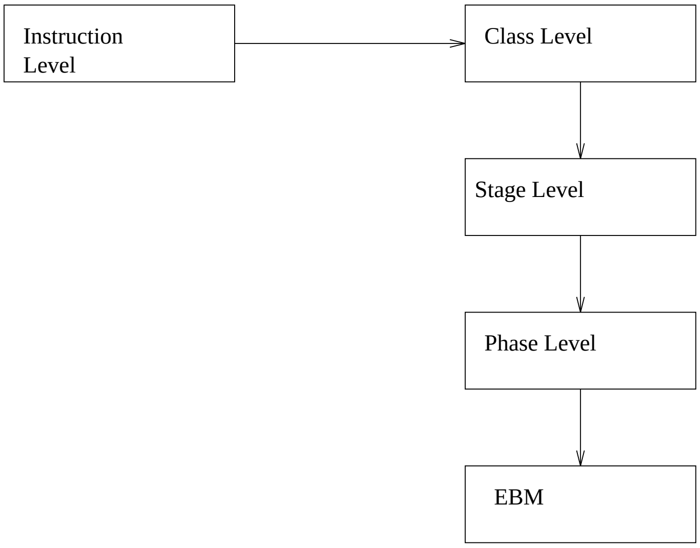

Figure 2.1: RISC Interpreter Model

execute in the exact same manner. The only thing that differentiates them is the opcode. So Tahar and Kumar created a Class Level which specifies instructions for only a few classes like ALU instructions. After a class level instruction has been verified, a specific instruction can be verified by instantiating a specific opcode.

Another important contribution of Tahar and Kumar is that they created a temporal abstraction function that handles the temporal abstraction happening as 5 instructions execute at the same time. Using this new temporal abstraction function and the hierarchy described in figure 2.1, Tahar and Kumar have verified the RISC core of their DLX-like processor.The verification of the lower levels in the hierarchy is done in a straight forward way. However, the proof that the stage level implies the Class level is more complex and undertaken in a different approach.

Tahar and Kumar split the overall verification into two parts:

- Assuming that some software constraints exist that prevent hazards, they prove:
  SW_Constraints, EBM $\vdash$ Correct_Instruction_Pipelining.

- If correct instruction pipelining can be assumed then the final goal can be realized:
  $EBM \vdash$ Instruction_level.

This research is important because it is the first attempt to come up with new techniques and methodologies to verify RISC processors.

2.5.3 Summary

In this section we presented the research that has already been done in the area of microprocessor verification. Most of the verified microprocessors listed here are very simple compared to modern microprocessors. Only two pipelined microprocessors have been completely verified: Mini Cayuga and Uinta. Of these two processors, only Uinta contains modern architectural features such as a five stage pipeline and data forwarding hardware. The verification presented in this thesis extends the verification techniques used in the Uinta proof to a more complex processor design.

Chapter 3

The EBM Specification of Sawtooth

3.1 The Sawtooth Microprocessor

SAWTOOTH is the successor microprocessor to Windley's AVM-1. It has a RISC-like architecture and has modern hardware features such as a five stage instruction pipeline, hardware for data forwarding, and hardware to deal with control hazards caused from branches and jumps. The circuit diagram for SAWTOOTH is shown in Figure 3.1.

SAWTOOTH has 48 general purpose registers each of which is a full 32 bits wide. Sixteen registers are freely available, while the other 32 registers are divided into two pages of 16. One of these pages is for supervisory mode registers while the other supplies general purpose registers. At any given time only 32 registers are visible to the programmer with a page address bit delegating the additional register page depending on whether or not the processor is in supervisory mode. A couple of these 48 registers are reserved. Register R0 is read-only and contains the value zero. Register R1 always holds a delayed value of the program counter. The processor also has a program status word which contains various status flags for the processor. The format for the program status word is shown in Table 3.1.

Table 3.1: Program Status Word

Page	SPVMode	AllowInterrupts	Carry	Overflow
4	3	2	1	0

Currently, 41 instructions have been incorporated into the design of the microprocessor.[1] The instructions use one of four opcode formats as shown in Figure 3.2. For more information concerning these instructions please see appendix B.

SAWTOOTH uses a five stage pipeline very similar to the pipeline shown for DLX by Hennessy and Patterson [HP90]. Each stage of the pipeline takes one clock cycle to execute, and each clock cycle

[1]The only instructions that have not been implemented are ALU instructions dealing with multiplication and division and the coprocessor instructions.

Table 3.2: Pipeline Structure

Stage	ALU instruction	Load or store instruction	Branch instruction
IF	IR←Mem[PC1]; PC1←PC1+4;	IR←Mem[PC1]; PC1←PC1+4;	IR←Mem[PC1]; PC1←PC1+4;
ID	Alu1←Rs1; Alu2←Rs2;	Alu1←Rs1; Alu2←Rs2; B←Rd ;	Alu1←Rs1; Alu2←Imm; A← Ra; B←Rd ;
EX	ALUout←Alu1 op Alu2;	ALUout← Alu1 + Alu2	ALUout←Alu1 + Alu2 COND←LOGIC(A,B);
MEM	ALUout1←ALUout	IR ← NOP ; Mem[AlUout]←SMDR; **or** ALUout1←Mem[ALUout]	if (COND) PC1←ALUout;
WB	Rd←ALUout1;	Rd←ALUout1;	NOP

is broken down into two phases. The designer took advantage of this fact and eliminated one level of data forwarding by performing the register write-back in stage five in the first phase and performing the register fetch in stage two in the second phase. This permits a register write-back and a register fetch to take place in the same clock cycle.

The actions that take place in the pipeline for typical ALU, Load and Store, and Branch instructions is shown in Table 3.2.

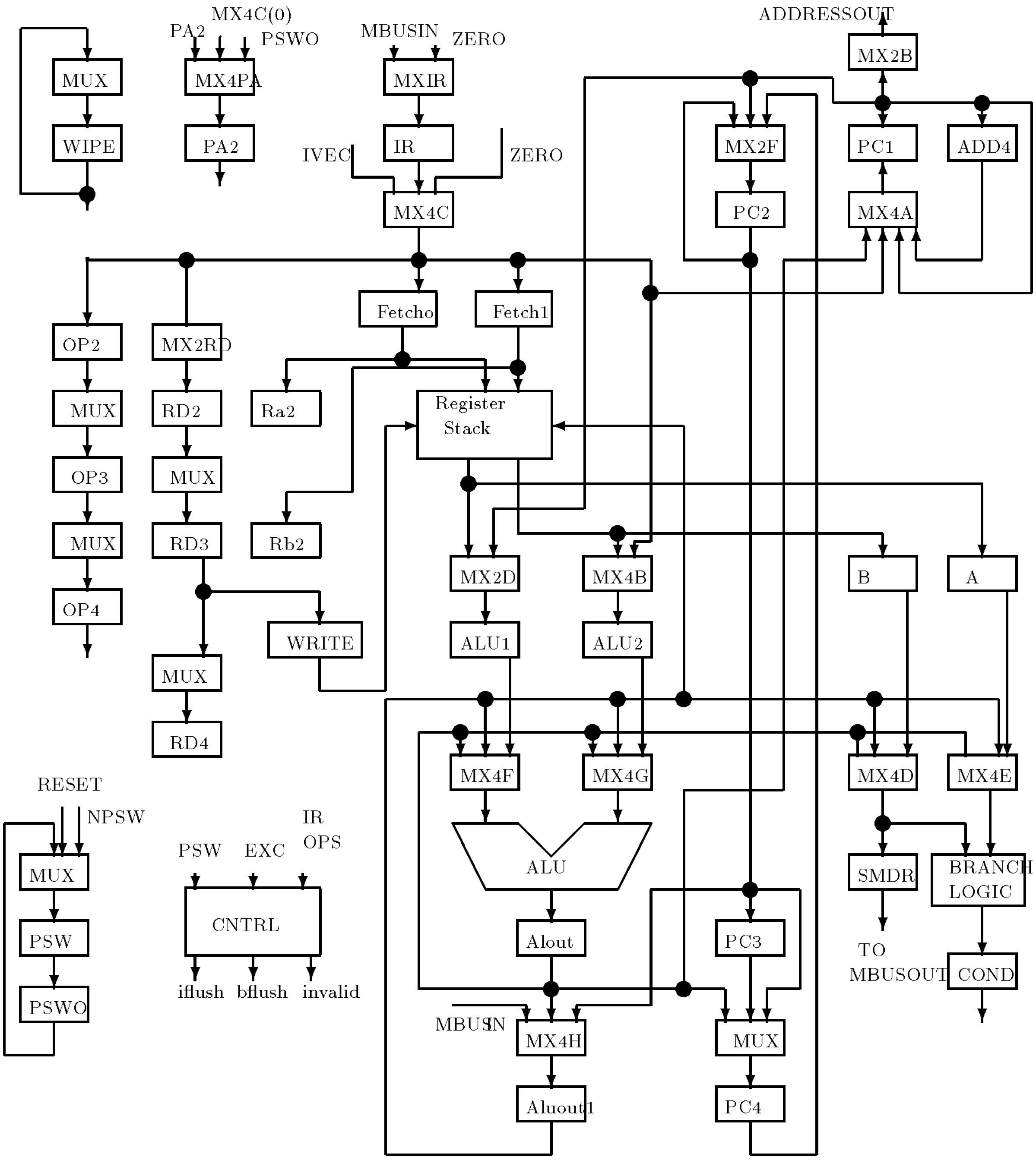

Figure 3.1: Circuit Diagram for Sawtooth

Opcode	Rd	Rs1	Rd2/Nu	Not Used
6 bits	5 bits	5 bits	5 bits	11 bits

Format 1

Opcode	Rd	Rs1/Nu	Immediate/NU
6 bits	5 bits	5 bits	16 bits

Format 2

Opcode	Rs2	Rs1/NU	Immediate/NU
6 bits	5 bits	5 bits	16 bits

Format 3

Opcode	Immediate/NU
6 bits	26 bits

Format 4

Figure 3.2: Opcode formats

3.2 The Abstract Representation

To deal with the very low level operations on bit strings we define an abstract representation based on uninterpreted types very much like the one used by Joyce [Joy89] and Windley [Win90b]. The abstract representation is defined as shown by Windley[Win94].

In our specification of Sawtooth, we use several abstract representations. An abstract representation is an abstract data type which defines several functions over uninterpreted types. For instance the abstract function `bit` takes a number n and a word w and returns the boolean value of the n-th bit of w. The type for the function `bit` is `:num -> *wordn -> bool`. The type `*wordn` is an uninterpreted type representing a word. For SAWTOOTH we define two abstract representations: one for operations dealing with bit strings, and one for the representation of memory. The abstract representation for bit string operations is shown in Table A.1 in appendix A. In this abstract representation we define several types which are used throughout our specifications:

- `*wordn`: An n-bit word. This represents the size of word in memory or the width of a register. We do not specify a concrete value for n.

- `*word5`: A bit string of length 5.

- `*word6`: A bit string of length 6. All opcodes and register indexes are of type `*wordn6`.

- `*word6->*wordn`: This type represents a function from a bit string of length 6 to a n-bit word. We use this type to describe the register file which uses a six bit register index to look up n-bit registers.

Using these type we can now define the state of the microprocessor at the electronic block model.

3.3 The EBM State

For the EBM level the information that compromises the state of SAWTOOTH is shown in Table 3.3. The state is defined using records as shown by Windley[Win94, page 5].

State Element	Description	Type
reg	Register File	`:*word6->*wordn`
psw	Program Status Word	`:*wordn`
pc1	Program counter	`:*wordn`
pc2	Program counter stage 2	`:*wordn`
pc3	Program counter stage 3	`:*wordn`
pc4	Program counter stage 4	`:*wordn`
mem	Memory	`:*memory`
alu1	Alu source register	`:*wordn`
alu2	Alu source register	`:*wordn`
aluout	Register that holds ALU output	`:*wordn`
aluout1	Holds previous value of aluout	`:*wordn`
a	Source register for branch logic	`:*wordn`
b	Source register for branch logic	`:*wordn`
ir	Instruciton Register	`:*wordn`
smdr	Store-Memory Data Register	`:*wordn`
pswo	A copy of previous psw	`:*wordn`
Cond	The condition code	`:bool`
ra2	Flip-flop holding source register	`:*word6`
rb2	Flip-flop holding second source register	`:*word6`
op2	Opcode of instruction in stage five	`:num`
op3	Opcode of instruction in stage five	`:num`
op4	Opcode of instruction in stage five	`:num`
rd2	Destination register for stage three	`:*word6`
rd3	Destination register for stage four	`:*word6`
rd4	Destination register for stage five	`:*word6`
pa2	Page address bit	`:bool`
wipe	Pipeline wipe flag	`:bool`
clk	Bool counter representing the clock	`:bool`

Table 3.3: The State of the Microprocessor at the EBM Level

The state is defined as an abstract data type with the elements shown in Table 3.3. If we wish access one element of the state we have to use a selector function. For example if we have a variable `s` of the type `:EBM_state` and we wish to access the state element `pc1` we use the selector function, `Pc1e` [2]. So to get the value of the `pc1` program counter we would write: `(Pc1e s)`.

3.3.1 The Environment

We also define the environment of the processor in the same manner as we did the state. The environment of the microprocessor consists of signals that come into the processor from external sources which affect

[2] The `e` at the end of `Pc1e` denotes that the selector function is one that works only for the EBM state.

the execution of instructions. For example, the `reset` flag is an external flag signalling the microprocessor to return to an initial state.

The environment of SAWTOOTH consists of the following:

- `reset`: The reset signal. Type: `:bool`.
- `ivec`: The interrupt vector. Type: `:*wordn`.
- `exc`: The exception flag. Type: `:bool`.

3.4 Basic Blocks

We begin the formal specification by defining the basic blocks from which we specify the behavior of larger blocks. The HOL definitions for a register, a flip-flop, and the various muxes we use are shown below. Most of these definitions are taken the original verification of AVM-1[Win90b].

```
MUX2_SPEC = ⊢def ∀ctl a b c. MUX2_SPEC ctl a b c = (c = (ctl →  a | b))


MUX3_SPEC =
⊢def ∀ctl1 ctl2 a b c d.
    MUX3_SPEC ctl1 ctl2 a b c d = (d = (ctl1 →  a | (ctl2 →  b | c)))


MUX4_SPEC =
⊢def ∀ctl1 ctl2 ctl3 a b c d e.
    MUX4_SPEC ctl1 ctl2 ctl3 a b c d e =
    (e = (ctl1 →  a | (ctl2 →  b | (ctl3 →  c | d))))


FF_SPEC =
⊢def ∀in ld q. FF_SPEC in ld q = (∀t. q(t + 1) = (ld t →  in t | q t))


REG_SPEC =
⊢def ∀i ld out contents.
    REG_SPEC i ld out contents =
    (∀t.
      (contents(t + 1) = (ld t →  i t | contents t)) ∧ (out = contents))
```

3.5 Specification Method

As opposed to previous verification efforts [Coh88b, Joy89] where the specification of the design was done after the chip was designed, here the specification for SAWTOOTH was done by a hand translation of the designer's behavioral VHDL for the microprocessor as the designer was finishing the design. Our goal was to have the **HOL** specification look as much like the VHDL as possible. For example consider the VHDL process shown below.

```
do_irtemp : PROCESS (op3, bflush, mbusin, mx4c(31 DOWNTO 26))
  BEGIN
  IF (op3 = "100000" OR op3="100010" OR bflush = '1') THEN
    -- LD or ST instruction or a BRANCH instruction taken
    irtemp <= "00000000000000000000000000000000";
  ELSIF (mx4c(31 DOWNTO 30) = "11" OR
         mx4c(31 DOWNTO 26) = "000101" OR
         mx4c(31 DOWNTO 26) = "000110") THEN
        -- BRANCH, JUMP, INT, RTI
    irtemp <= "00000000000000000000000000000000";
  ELSE
    irtemp <= mbusin;
  END IF;
  END PROCESS do_irtemp;
```

The process *irtemp* is the multiplexor that outputs the new value for the instruction register. The process inserts a NOOP instruction here if the instruction executing in the forth pipeline stage is a LOAD or a STORE. This is done because there is only one bus to memory, so an instruction fetch here would cause a structural hazard.

The process inserts a NOOP if the instruction being decoded in stage two is a branch, jump, interrupt, or a return from interrupt. The designer does this as a primitive form of branch prediction. After the NOOP, the instruction immediately after the branch is fetched as if the branch were not taken. The NOOP is inserted here to make sure that no state updating actions occur that can not be undone.

If none of these conditions occur then the mux passes the value being fetched from memory. Our **HOL** specification for this process is shown below.[3]

```
MXIR_SPEC =
⊢def ∀bflush op op3 orep pc1 mbus out.
    MXIR_SPEC orep(bflush,op3,op,pc1,mbus,out) =
    (∀t.
      let ctl1 = (op3 t) IN {32,34} ∨ bflush t
      in
       let ctl2 = (op t) IN {5,6,56,57,54,48,51,50,53,52,49,55}
       in
        let a = wordn orep 0
        in
         let b = mbus t in MUX3_SPEC ctl1 ctl2 a a b(out t))
```

The **HOL** specification for this process uses the basic block definition for a three input multiplexor: `MUX3_SPEC`. Following the definition of `MUX3_SPEC`, the specification states that:

- If the condition `ctl1` is true then the result `a` is forwarded. That is, if the instruction in stage four is a LOAD or a STORE or a branch flush condition exists, then a NOOP is forwarded into the instruction register.

[3] To specify checking for certain opcodes we use the sets notation. Unfortunately the numbers that are contained in the **HOL** sets do not match the numbers of the instruction opcodes listed in Appendix B. Readers should note that the numbers in the **HOL** sets are decimal numbers and the opcode values listed in Appendix B are octal values.

- If the condition `ctl2` is true, then once again a NOOP is output. That is, if the instruction being decoded in stage two is a branch, a JUMP, JALI, INT, or RTI, then a NOOP is inserted into the instruction register.
- If neither of the two conditions hold, then data from the result `b` is output. That is the value of the instruction fetched from memory is output.

In the complex multiplexor specifications, the logical conditions that control the output will be defined by the `ctl` let statements in the specifications. The results output by the mux will often be represented place holder names like `a` and `b` above.

Throughout this report we will present the VHDL process definitions alongside our **HOL** definitions to provide a basis for comparison.

3.6 The Data Path

3.6.1 The Instruction Register Section

The *IR section* of SAWTOOTH consists of the instruction register, the mux that loads the IR, the mux that decodes the instruction word, the muxes that select the inputs into the register file, and the register file itself. The circuit diagram for the part of SAWTOOTH we will be specifying in this section is shown in Figure 3.3.

Mux MX4C

We already presented the specification for the MXIR multiplexor in section 3.5. Probably the most important block in this diagram is the mux MX4C which selects either the contents of the instruction register, a NOOP instruction, or an interrupt instruction as the next instruction to be decoded. This mux is also where our specification differs the most from the VHDL. In the VHDL the process definition does what we just described; the MX4C selects one of three possible instruction words to be decoded. The VHDL for the process is shown below.

```
do_mx4c : PROCESS (wipe, invalid, bflush, iflush, ivec, ir)
  BEGIN
  IF (iflush = '1') THEN
    mx4c(31 DOWNTO 8) <= "000101000100000000000000";
    mx4c(7 DOWNTO 0) <= ivec;
    -- INT r1, r0 immediate instr.
  ELSIF (wipe = '1' OR invalid = '1' OR bflush = '1') THEN
    mx4c <= "00000000000000000000000000000000";
  ELSE
    mx4c <= ir;
  END IF;
  END PROCESS do_mx4c;
```

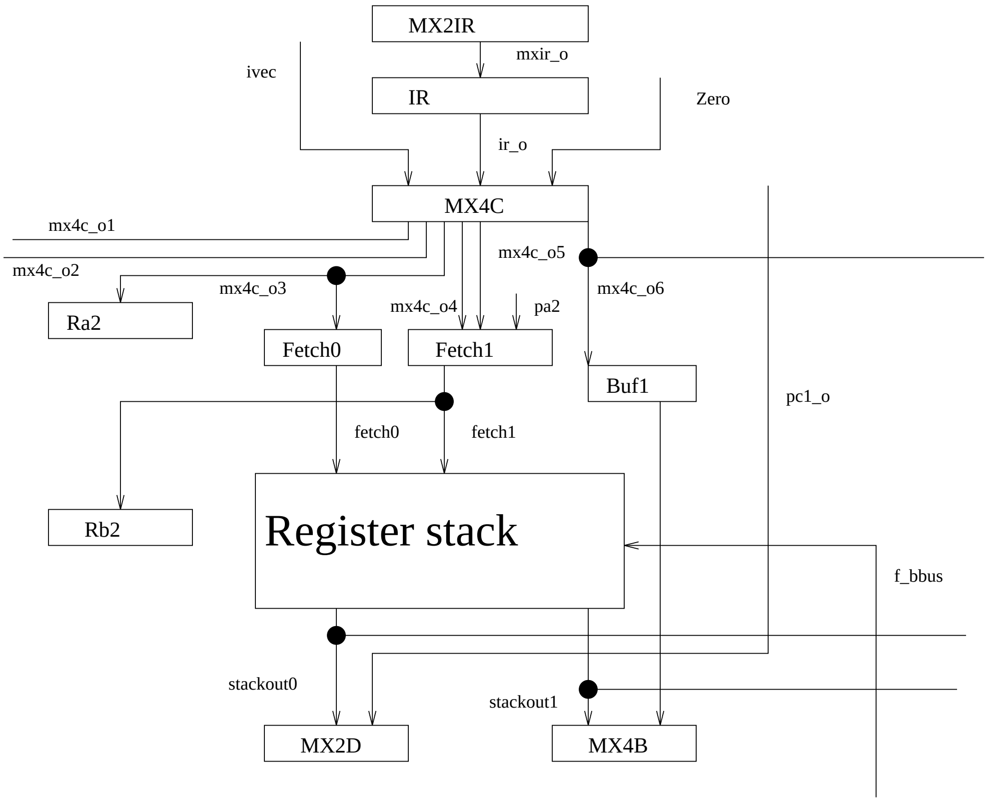

Figure 3.3: Instruction Decode section of SAWTOOTH

The VHDL states that if an interrupt flush [4] condition exists then an INT instruction is inserted into the pipeline at stage two. If a wipe condition exists, or the instruction being decoded is invalid, or a branch flush condition exists, then a NOOP is inserted into the pipeline. Otherwise the mux outputs the value of the instruction register.

In our specification we carry the process one step farther. We specify MX4C to output six items representing the decoded parts the selected instruction word. The main reason for this is that eventually we must do a type conversion from a `*wordn` instruction to the various parts of a decoded word. We use the the definitions `Wordn_decode` and `INST_Decode`, which are defined in Appendix C, to perform the type conversion. The six parts of the decoded word which are forwarded are:

- Opcode–op. Type: `num`.
- Destination Register–rd. Type: `*word5`.
- Source Register 1–ra. Type: `*word5`.
- Source Register 2–rb. Type: `*word5`.
- 16 bit Immediate value–imm. Type: `*short`.

[4] The control lines `iflush` and `bflush` are explained in section 3.7.

- 26 bit Immediate value–imm26. Type: `*word26`.

A single instruction does not contain all of these. For example the only instruction that contains the 26 bit immediate value is the JUMP instruction.

We chose to place this type conversion the specification of `MX4C` so that every block dependent upon the output of `MX4C` would not have to perform the same actions. The specification of the `MX4C` mux is presented below.

```
MX4C_SPEC =
⊢def ∀wipe invalid orep bflush iflush decrep ivec ir out1 out2 out3 out4
    out5 out6.
    MX4C_SPEC orep decrep
    (wipe,invalid,bflush,iflush,ivec,ir,out1,out2,out3,out4,out5,out6) =
    (∀t.
      out1 t,out2 t,out3 t,out4 t,out5 t,out6 t =
      (iflush t →
       INST_Decode(INT R1 R0(low16w orep(ivec t))) |
       ((wipe t ∨ invalid t ∨ bflush t) →
        INST_Decode NOOP |
        ((ir t = wordn orep 0) →
         INST_Decode NOOP |
         Wordn_decode decrep(ir t)))))
```

It should be noted that with the combination of the mux `MXIR` and the mux `MX4C`, Sawtooth uses a *2-point decision procedure* to decode instructions. Just because, the instruction fetched into the instruction register, `ir`, is an ADD, does not necessarily mean that the ADD instruction will execute through all five stages of the pipeline.

Fetch0 and the Fetch1

The `FETCH0` and `FETCH1` muxes forward two 6-bit register addresses to the register file. These addresses are used to fetch the values of the two source operands for the ALU.

The `FETCH1` logic is actually a multiplexor which selects the correct value for the `rb2` flip-flop. This value is also forwarded to the register file on the line `fetch1`. The multiplexor selects between forwarding the value of the destination register, `rd`, and that of the second source register `rb`. If the instruction is a load or branch operation then the destination register is forwarded, otherwise the the value of `rb` is selected.

```
do_fetch1 : PROCESS (mx4c(31 DOWNTO 11),pa2)
  BEGIN
  IF (mx4c(31 DOWNTO 26) = "100010" OR mx4c(31 DOWNTO 29)="110") THEN
     -- Branch or ST Instr.
    IF (pa2 = '0') THEN
      fetch1(5) <= pa2;
      fetch1(4 DOWNTO 0) <= mx4c(25 DOWNTO 21);
    ELSE
      fetch1(5) <= mx4c(25);
      fetch1(4) <= '0';
      fetch1(3 DOWNTO 0) <= mx4c(24 DOWNTO 21);
    END IF;
  ELSE
    IF (pa2 = '0') THEN
      fetch1(5) <= pa2;
      fetch1(4 DOWNTO 0) <= mx4c(15 DOWNTO 11);
    ELSE
      fetch1(5) <= mx4c(15);
      fetch1(4) <= '0';
      fetch1(3 DOWNTO 0) <= mx4c(14 DOWNTO 11);
    END IF;
  END IF;
  END PROCESS do_fetch1;
```

The **HOL** specification for this process looks rather convoluted. This is because the mux attaches the value in the page address flip-flop, `pa2`, to the selected register index to create a six bit address that the register file needs to address a specific register. The page address bit denotes which two pages of registers are visible to the program in execution.

In both the `FETCH0` and `FETCH1` processes the six-bit [5] register address is computed by following the steps listed below. Since both the VHDL and **HOL** specifications for this process are rather complex we will break down the first case.

- If the page address bit is low then the registers visible to the user are register R0 through R31 (as shown in Figure 3.4). To create the six-bit register address:
  - The sixth bit is set to 0.
  - The remaining 5 bits are set to the original register address. Having a 0 in the high order bit of the address ensures that only registers R0-R31 can be accessed.
    * In VHDL this is written as:

```
fetch0(5) <= pa2 ;
fetch0(4 DOWNTO 0) <= mx4c(20 DOWNTO 16);
```

    * In **HOL** this is written as:

```
setb6 orep(5,extend5 orep(rd t),pa2 t)
```

[5] Note that in a 32-bit word, bits are numbered starting at 0.

    * The **HOL** specification says to extend the five-bit register address to a six-bit address and then set the high-order bit to the value of the page address bit.

- If the page address bit is high, then registers R0-R15 and R32-R47 are visible to the user. In this case:
  - The sixth bit is set to the high order bit in the original 5 bit address.
  - The fifth bit of the new address is set to 0.
  - The last four bits of the new address are just the four low order bits of the original five-bit address. This switches register addresses in the range 16 through 31 to the range 32 to 47.

```
FETCH1_SPEC =
⊢def ∀op rd orep out pa2 rb.
    FETCH1_SPEC orep(op,rd,rb,pa2,out) =
    (∀t.
      let temp = extend5 orep(rd t)  in
       let temp2 = extend5 orep(rb t)  in
        let a = bit5 orep 5(rd t)  in
         let c = bit5 orep 5(rb t)
         in
          (out t =
           (((op t) IN {54,48,51,50,53,52,49,55} ∨ (op t) IN {34}) →
            ((¬pa2 t) →
             setb6 orep(5,extend5 orep(rd t),pa2 t) |
             setb6 orep(5,setb6 orep(4,temp,F),a)) |
            ((¬pa2 t) →
             setb6 orep(5,extend5 orep(rb t),pa2 t) |
             setb6 orep(5,setb6 orep(4,temp2,F),c)))))
```

The FETCH0 logic just forwards the value of the first source register to the register file. It also extends the five-bit register address from the opcode to a full six-bit register address needed by the register file.

```
do_fetch0 : PROCESS (mx4c(20 DOWNTO 16), pa2)
  BEGIN
  IF (pa2 = '0') THEN
    fetch0(5) <= pa2;
    fetch0(4 DOWNTO 0) <= mx4c(20 DOWNTO 16);
  ELSE
    fetch0(5) <= mx4c(20);
    fetch0(4) <= '0';
    fetch0(3 DOWNTO 0) <= mx4c(19 DOWNTO 16);
  END IF;
  END PROCESS do_fetch0;
```

```
FETCH0_SPEC =
⊢def ∀pa2 ra out orep.
    FETCH0_SPEC orep(pa2,ra,out) =
    (∀t.
      out t =
      (let temp = ra t   in
        let temp2 = extend5 orep(ra t)
        in
         ((¬pa2 t) →
          setb6 orep(5,temp2,pa2 t) |
          setb6 orep(5,setb6 orep(4,temp2,F),bit5 orep 5(ra t)))))
```

The Register File

The register file contains 48 general purpose registers each 32 bits in length, but at any given time only 32 registers are visible to a process. The registers are organized into three pages of sixteen registers each. The first page, which holds the registers 0-15, are common to all processes. However the page address bit specifies which of the remaining two pages of registers is seen by the executing process. If the page address bit is 0 then the registers 16-31 are the second page visible to the process. Otherwise the registers 32-47 are the second set of sixteen available to the process. This relationship between the page address bit and what registers are used is shown in Figure 3.4.

For each clock cycle, the register file must update the destination register for the instruction executing in the write back stage–`rd4`. The destination register, `rd4`, is updated with the value residing in the `aluout1` register. The write back to the register file is done in the first phase of the clock cycle. During the second phase of the clock cycle the register file inputs the two source registers of the current instruction being executed. These values come in on the lines `fetch0` and `fetch1`. Using the values for `ra` and `rb`, the register files outputs register contents of the two source registers. These two words are forwarded on the lines `stackout0` and `stackout1`. Executing the write back one phase earlier than the register fetch allowed the designer to eliminate one level of data forwarding.

We model the register file as a function of type `:*word6 -> *wordn`. That is, the register file is a function that takes a source register or type `*word6` and returns the contents of that register.

We use the definition `UPDATE_REG` (see Appendix C) to model the write back operation.

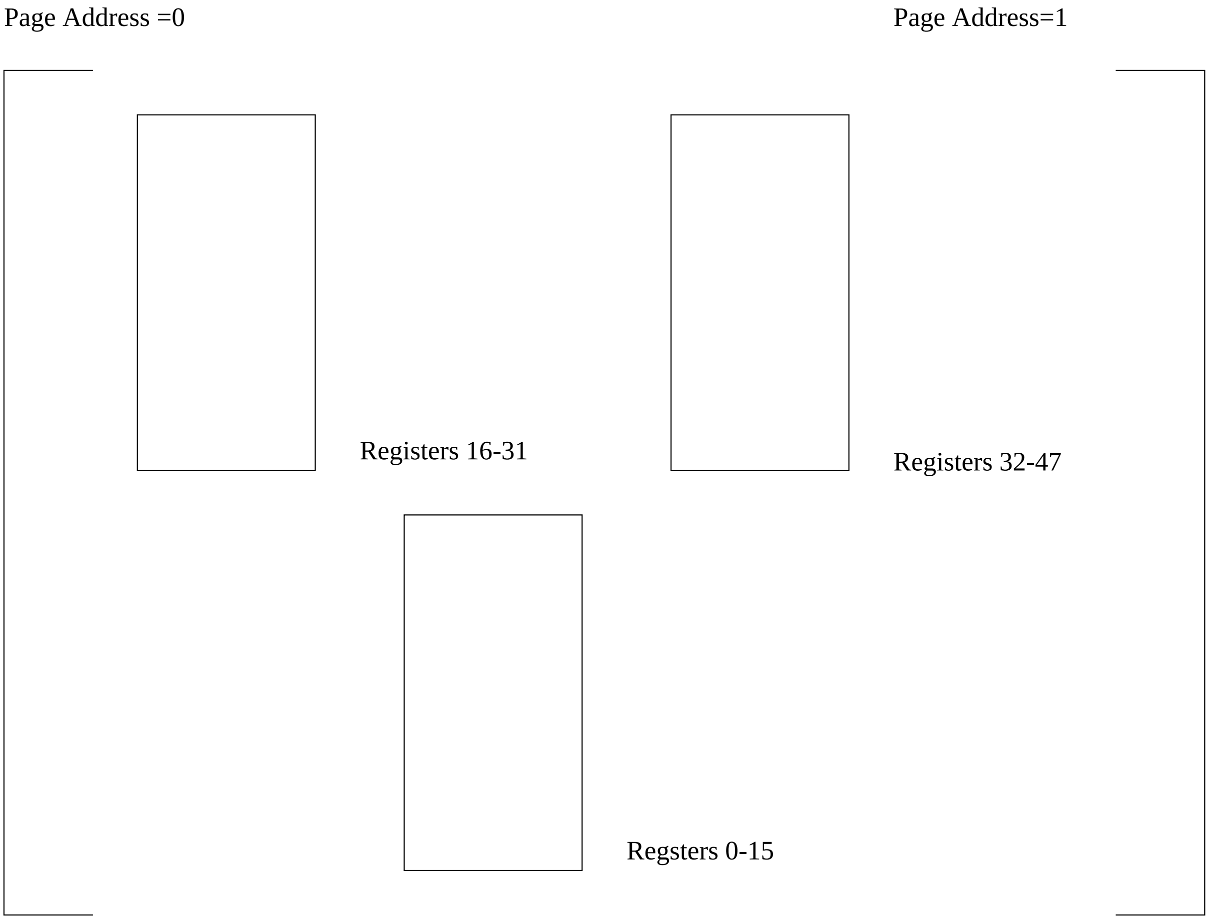

Figure 3.4: The Page Address Bit and Registers

```
REGISTER_BLOCK =
⊢def ∀orep fetch0 fetch1 regin stackout0 stackout1 clk_1 clk_2 rd4 regfile.
    REGISTER_BLOCK
    orep
    (clk_1,clk_2,fetch0,fetch1,regin,rd4,stackout0,stackout1,regfile) =
    (∀t.
      (regfile(t + 1) = (clk_1 t → UPDATE_REG orep(rd4 t)(regfile t)(regin t) | regfile t)) ∧
      (clk_2 t ⟹ (stackout0 t = regfile t(fetch0 t))) ∧
      (clk_2 t ⟹ (stackout1 t = regfile t(fetch1 t))))
```

Muxes MX2D and MX4B

The muxes MX2D and MX4B forward the correct value to the ALU source registers. The multiplexor MX2D outputs either the contents of the first source register, from `stackout0`, or if the instruction is a branch then the mux forwards the value of the program counter so the ALU can calculate the branch target address.

In the VHDL the actions of this multiplexor are defined in the the process `alu1` shown below.

```
do_alu1 : PROCESS (clk)
  BEGIN
  IF (clk='1' AND clk'LAST_VALUE='0') THEN
    IF( mx4c(31 DOWNTO 29) = "110") THEN -- Branch Instructions
      alu1 <= pc1;
    ELSE
      alu1 <= stackout0;
    END IF;
  END IF;
  END PROCESS do_alu1;


MX2D_SPEC =
⊢def ∀op pc1 stackout0 out.
    MX2D_SPEC(op,pc1,stackout0,out) =
    (∀t.
      let ctl1 = (op t) IN {54,48,51,50,53,52,49,55}
      in
       MUX2_SPEC ctl1(pc1 t)(stackout0 t)(out t))
```

Mux MX4B outputs the second value to be used as input to the ALU. The value that this mux outputs is stored in the register `alu2`. The value held in the `alu2` register is then forwarded into the data forwarding registers as one of the possible operands for the ALU.

If the instruction is a ALU immediate instruction (i.e. ADDI), then the 16 bit immediate from our decoded word is extended to 32 bits and outputted. If the instruction is a LOAD, STORE, or LHI then the sign extension of the immediate is forwarded. If the instruction is a branch, JUMP, or JALI then the sign extended immediate value is also forwarded. If none of these conditions hold, the contents of the second source register, `rb2`, is outputted.

```
MX4B_SPEC =
⊢def ∀op imm stackout1 orep out.
    MX4B_SPEC orep(op,imm,stackout1,out) =
    (∀t.
      let ctl1 = (op t) IN {8,9,10,11,12,13,14,1,2,3,4,5,6} in
      let ctl2 = (op t) IN {32,34,33} in
      let ctl3 = (op t) IN {54,48,51,50,53,52,49,55} ∨ (op t) IN {56,57}
        in
         let a = extend orep(imm t)  in
          let b = signextend orep(imm t) in
           MUX4_N_SPEC ctl1 ctl2 ctl3 a b b(stackout1 t)(out t))
```

In the VHDL the logic for this mux is defined in the process `alu2` shown below.

```
do_alu2 : PROCESS (clk)
  BEGIN
  IF (clk='1' AND clk'LAST_VALUE='0') THEN
    IF (mx4c(31 DOWNTO 29) = "100") THEN -- LD, ST, or LHI instruction
      IF(mx4c(15) = '1') THEN
        alu2(31 DOWNTO 16) <= "1111111111111111";
      ELSE
        alu2(31 DOWNTO 16) <= "0000000000000000";
      END IF;
      alu2(15 DOWNTO 0) <= mx4c(15 DOWNTO 0);
    ELSIF (mx4c(31 DOWNTO 30) = "00" ) THEN
          -- Special Instructions or Immediate ALU instructions
      alu2(31 DOWNTO 16) <= "0000000000000000";
      alu2(15 DOWNTO 0) <= mx4c(15 DOWNTO 0);
    ELSIF (mx4c(31 DOWNTO 30) = "11") THEN -- Branch,Jump, or JALI
      IF(mx4c(15) = '1') THEN
        alu2(31 DOWNTO 16) <= "1111111111111111";
      ELSE
        alu2(31 DOWNTO 16) <= "0000000000000000";
      END IF;
      alu2(15 DOWNTO 0) <= mx4c(15 DOWNTO 0);
    ELSE
      alu2 <= stackout1;
    END IF;
  END IF;
  END PROCESS do_alu2;
```

3.6.2 The ALU Section

This section presents the specifications of the blocks in the ALU section of the processor. A circuit diagram for this area of the microprocessor is is shown in Figure 3.5. The ALU section consists of the ALU proper, the data forwarding logic, and the logic which loads the forwarding registers. Each of these are shown below.

The ALU

The ALU has two functions: it performs the desired arithmetic instruction and it computes the new program status word. Our specification encapsulates this idea with two outputs–one for the ALU result and one for the new program status word. An important item to notice about the following **HOL** definition is that for the two output values a check is added to enforce the property that these two values are updated in only the second phase of the clock cycle. The two outputs of the ALU are represented as the return values the two functions, `ALU_func` and `PSW_func`, [6] which along with the ALU specification are shown below.[7]

---

[6]The function `PSW_func` uses the **HOL** function `except` extensively. This behavior of this function is described in Appendix A.

[7]The VHDL code for the ALU is shown in Appendix E.

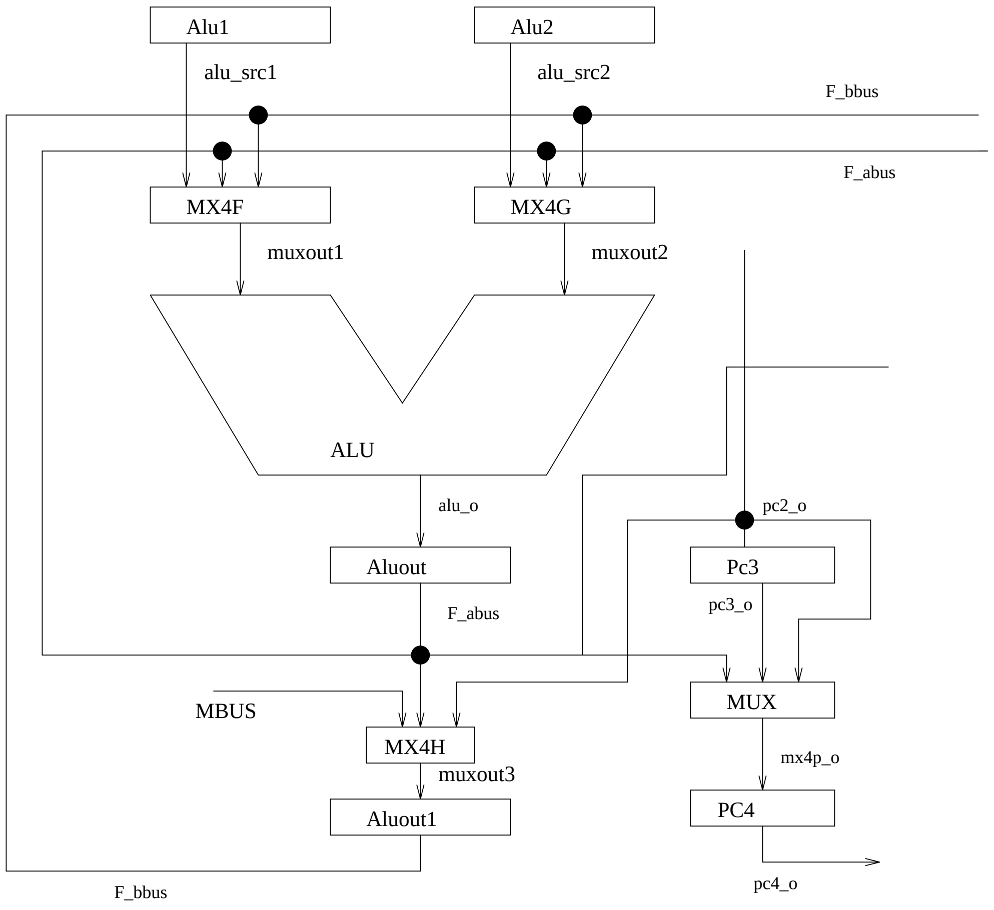

Figure 3.5: ALU section of SAWTOOTH

```
ALU_func =
⊢def ∀orep mx4f mx4g psw op carry.
  ALU_func orep op psw carry(mx4f,mx4g) =
  (op IN {16,8} → add orep(mx4f,mx4g) |
  (op IN {17,9} → addc orep(mx4f,mx4g,carry) |
  (op IN {18,10} → sub orep(mx4f,mx4g) |
  (op IN {19,11} → subc orep(mx4f,mx4g,carry) |
  (op IN {12,20} → band orep(mx4f,mx4g) |
  (op IN {13,21} → bor orep(mx4f,mx4g) |
  (op IN {14,22} → bxor orep(mx4f,mx4g) |
  (op IN {23} → bnot orep mx4f |
  (op IN {24} → shl orep(mx4f,1) |
  (op IN {25} → shr orep(mx4f,1) |
  (op IN {26} → setbit orep(msbv,shr orep(mx4f,1),bit orep msbv mx4f) |
  (op IN {27} → shl orep(mx4f,8) |
  (op IN {28} → shr orep(mx4f,8) |
  (op IN {29} → setrange orep(shr orep(mx4f,8))(31,24)
                (repeat orep(8,bit orep msbv mx4f)) |
  (op IN {2} → ((repeat orep(32,F)) except [lsbv,get_pa orep psw]) orep |
  (op IN {3} → psw |
  (op IN {5} → band orep(add orep(mx4f,mx4g),wordn orep 255) |
  (op IN {6} → add orep(mx4f,mx4g) |
  (op IN {54,48,51,50,53,52,49,55} → add orep(mx4f,mx4g) |
  (op IN {56,57} → add orep(mx4f,mx4g) |
  (op IN {32,34} → add orep(mx4f,mx4g) |
  (op IN {33} → combine orep (lower16 orep mx4g,lower16 orep mx4f) |
                 ARB))))))))))))))))))))))


PSW_func =
⊢def ∀orep mx4f mx4g psw op carry.
  PSW_func orep op psw carry(mx4f,mx4g) =
  (op IN {16,8} → (psw except [vf,aovfl orep(mx4f,mx4g);cf,addp orep(mx4f,mx4g)])orep |
  (op IN {17,9} → (psw except [vf,acovfl orep(mx4f,mx4g,bit orep cf psw); cf,addcp orep(mx4f,mx4g,bit orep cf psw)]) orep |
  (op IN {18,10} → (psw except [vf,sovfl orep(mx4f,mx4g);cf,subp orep(mx4f,mx4g)]) orep |
  (op IN {19,11} → (psw except [vf,scovfl orep(mx4f,mx4g,bit orep cf psw); cf,subcp orep(mx4f,mx4g,bit orep cf psw)]) orep |
  (op IN {24} → (psw except [cf,bit orep msbv mx4f])orep |
  (op IN {25,26} → (psw except [cf,bit orep lsbv mx4f])orep |
  (op IN {1} → (psw except [pa,bit orep lsbv mx4g])orep |
  ((op IN {4} ∧ bit orep sm psw) → mx4f |
  (op IN {5} → ((shl orep(psw,8)) except [pa,bit orep pa psw;sm,T;ie,F;cf,bit orep cf psw;vf,bit orep vf psw]) orep |
  (op IN {6} → setbit orep(26,shr orep(psw,8),T) | psw))))))))))


ALU_SPEC =
⊢def ∀orep mx4f mx4g psw out op alu out2 carry clk.
  ALU_SPEC orep(clk,mx4f,mx4g,psw,op,carry,alu,out,out2) =
  (∀t.
  (out t = (clk t → ALU_func orep(op t)(psw t)(carry t)(mx4f t,mx4g t) | ARB)) ∧
  (out2 t = (clk t → PSW_func orep(op t)(psw t)(carry t)(mx4f t,mx4g t) | ARB)))
```

Data Forwarding

Data forwarding is used to eliminate data hazards caused by the parallel execution of instructions in the pipeline. The ALU output of the last two instructions are kept in two registers, `aluout` and `aluout1`, and latched onto buses which carry the data to the muxes `MX4F` and `MX4G`. Using the values from `aluout` and `aluout1` and the values contained in `alu1` and `alu2`, the multiplexors `MX4F` and `MX4G` forward two operands to the ALU.

Mux `MX4F` selects either the contents of the `alu1` register or one of the results from the previous two instructions. If the instruction to execute in the ALU is a branch instruction then no forwarding is needed so the contents of the `alu1` register are forwarded. If `ra2` is the same as `rd3` (the destination register of the previous instruction) then the value in `aluout` is forwarded back into the ALU. If `ra2` is the destination register for the next to last instruction, then the contents of the `aluout1` register are forwarded. If none of these conditions hold then there is no data hazard and the contents of the `alu1` register are passed to the ALU.

```
do_mx4f : PROCESS (ra2, rd3, rd4, op2(5 DOWNTO 3), aluout, aluout1, alu1)
  BEGIN
  IF op2(5 DOWNTO 3) = "110" THEN -- If it's a BRANCH instr.
    mx4f <= alu1;
  ELSIF (ra2 = rd3 AND rd3(4 DOWNTO 1) /= "0000") THEN
    mx4f <= aluout;
  ELSIF (ra2 = rd4 AND rd4(4 DOWNTO 1) /= "0000") THEN
    mx4f <= aluout1;
  ELSE
    mx4f <= alu1;
  END IF;
  END PROCESS do_mx4f;


MX4F_SPEC =
⊢def ∀alu1 aluout aluout1 op2 out ra2 rd3 rd4 orep.
    MX4F_SPEC orep(alu1,aluout,aluout1,op2,ra2,rd3,rd4,out) =
    (∀t.
      let ctl1 = (op2 t) IN {54,48,51,50,53,52,49,55} in
       let ctl2 = (ra2 t = rd3 t) ∧ ¬(rd3 t = R0) in
        let ctl3 = (ra2 t = rd4 t) ∧ ¬(rd4 t = R0)
        in
  MUX4_SPEC ctl1 ctl2 ctl3(alu1 t) (aluout t) (aluout1 t)(alu1 t)(out t))
```

The mux `MX4G` works in almost exactly the same manner as `MX4F` except it considers the contents of the `rb2` source register which are held in the register `alu2`.

```
do_mx4g : PROCESS (rb2, rd3, rd4, aluout, aluout1, alu2, op2(5 DOWNTO 3))
  BEGIN
  IF (op2(5 DOWNTO 3) /= "010") THEN -- If its not an ALU instruction then
    mx4g <= alu2;
  ELSIF (rb2 = rd3 AND rd3(4 DOWNTO 1) /= "0000") THEN
    mx4g <= aluout;
  ELSIF (rb2 = rd4 AND rd4(4 DOWNTO 1) /= "0000") THEN
    mx4g <= aluout1;
  ELSE
    mx4g <= alu2;
  END IF;
  END PROCESS do_mx4g;



MX4G_SPEC =
⊢def ∀alu2 aluout aluout1 op2 out ra2 rd3 rd4 orep.
    MX4G_SPEC orep(alu2,aluout,aluout1,op2,ra2,rd3,rd4,out) =
    (∀t.
      let ctl1 = ¬(op2 t) IN {16,17,18,19,20,21,22,23} in
       let ctl2 = (ra2 t = rd3 t) ∧ ¬(rd3 t = R0) in
        let ctl3 = (ra2 t = rd4 t) ∧ ¬(rd4 t = R0)
        in
   MUX4_N_SPEC ctl1 ctl2 ctl3 (alu2 t) (aluout t) (aluout1 t) (alu2 t)(out t))
```

Mux MX4H

The multiplexer MX4H outputs the next value to be loaded into the register `aluout1`. If the instruction executing in stage four, `op3`, is an interrupt or a JALI instruction then the value of the contents of the `pc2` register are passed on to `aluout1`. If `op3` is a LOAD instruction then the word coming in from memory is outputted to the `aluout1` so that it may be immediately forwarded to the next instruction. If none of these conditions hold then the contents of the `aluout` register are outputted. The logic for this mux is located in the process `aluout1`.

```
do_aluout1 : PROCESS (clk)
  BEGIN
  IF (clk='1' AND clk'LAST_VALUE='0') THEN
    IF (op3 = "111001" OR op3 = "000101") THEN -- INT or JALI instr.
      aluout1 <= pc2;
    ELSIF (op3 = "100000") THEN -- LD instr.
      aluout1 <= mbusin;
    ELSE
      aluout1 <= aluout;
    END IF;
  END IF;
  END PROCESS do_aluout1;


MX4H_SPEC =
⊢def ∀mbus aluout muxin1 op pc2 mbusin out.
    MX4H_SPEC(mbusin,aluout,muxin1,op,pc2,out) =
    (∀t.
      let ctl1 = (op t) IN {5,57}
      in
       let ctl2 = (op t = 32)
       in
        MUX3_N_SPEC ctl1 ctl2(pc2 t)(mbusin t)(aluout t)(out t))
```

The Carry Logic

Although it is not shown in the circuit diagram there is a separate logic which calculates the carry bit for the current instruction and forwards this value to the ALU. The carry flag is set based on the instruction which is about to execute in the ALU. The **HOL** function `get_cf` returns the value of the carry flag in the program status word.

```
do_carry : PROCESS(op2,psw(1))
  BEGIN
  IF (op2(5 DOWNTO 3) = "001" OR op2(5 DOWNTO 3) = "010") THEN -- ALU and ALUI
    CASE op2(2 DOWNTO 0) IS
      WHEN "000" => -- ADD and ADDI
        carry <= '0';
      WHEN "001" => -- ADDC and ADDCI
        carry <=  psw(1);
      WHEN "010" => -- SUB and SUBI
        carry <= '1';
      WHEN "011" => -- SUBC and SUBCI
        carry <=  psw(1);
      WHEN OTHERS => -- Must have received X's or Z's as inputs
        carry <= '0';
    END CASE;
  ELSE
    carry <= '0';
  END IF;
  END PROCESS do_carry;
```

```
CARRY_LOGIC_SPEC =
⊢def ∀op psw orep out.
    CARRY_LOGIC_SPEC orep(op,psw,out) =
    (∀t.
      out t =
      ((op t) IN {16,8} →    F |
       ((op t) IN {17,9} →    get_cf orep(psw t) |
        ((op t) IN {18,10} →    T |
         ((op t) IN {19,11} →  get_cf orep(psw t) | F)))))
```

3.6.3 The PC Section

The PC section of the microprocessor includes the four registers that hold the program counter values for four of the instructions in the pipeline. It also contains muxes that select the new values of the various program counters.

MX4A

The mux `MX4A` outputs the new value of the `pc1` register. If the instruction executing in the fourth stage is a branch, INT, RTI, or JALI, the mux forwards the branch target address calculated by the ALU. If the instruction being decoded in `MX4C` is a JUMP then the mux outputs the value of the 26 bit immediate extended to 32 bits. If the instruction in stage four is a LOAD or STORE then `MX4C` has inserted a NOOP instruction and the mux outputs the present value of the program counter. If none of these conditions hold then the mux outputs the program counter incremented. This actions of this multiplexor are shown in the process `pc1` in the VHDL.

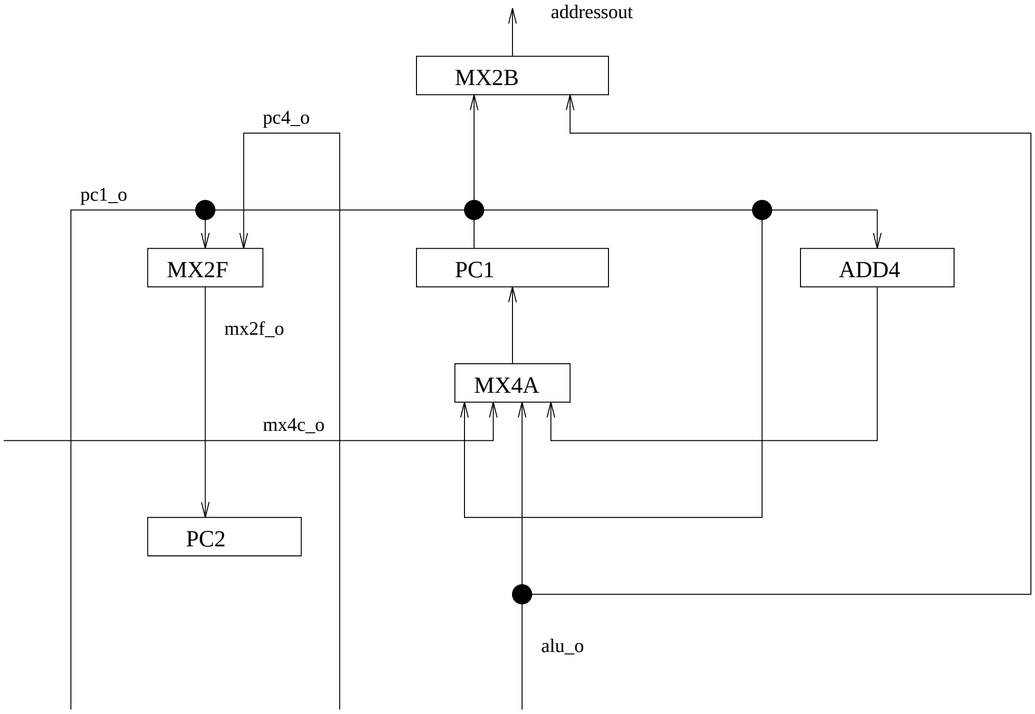

Figure 3.6: The Program Counter Section

```
do_pc1 : PROCESS (clk)
  VARIABLE temp : qsim_state_vector (16 DOWNTO 0);
  VARIABLE tempint :integer;
  BEGIN
  IF (clk='1' AND clk'LAST_VALUE='0') THEN
    IF (op3 = "000101" OR op3 = "000110" OR op3 = "111001") THEN
       -- INT, RTI, or JALI Instr.
      pc1 <= aluout;
    ELSIF (op3(5 DOWNTO 3) = "110" AND cond = '1') THEN
          -- Branch instruction was TAKEN
      pc1 <= aluout;
    ELSIF mx4c(31 DOWNTO 26) = "111000" THEN
          -- A JUMP Instr.
      pc1(31 DOWNTO 26) <= "000000";
      pc1(25 DOWNTO 0) <= mx4c(25 DOWNTO 0);
    ELSIF op3 = "100000" OR op3 = "100010" THEN
          -- LD, or ST
      NULL; -- Don't load pc1
    ELSIF (mx4c(31 DOWNTO 30) = "11" OR
           mx4c(31 DOWNTO 26) = "000101" OR
           mx4c(31 DOWNTO 26) = "000110") THEN
          -- BRANCH, JUMP, INT, RTI
      NULL; -- Don't load pc1
    ELSE
      temp := to_qsim_state(to_integer('0'&pc1(15 DOWNTO 0),0)+1,17);
      tempint := to_integer('0'&temp(16),0);
      pc1(15 DOWNTO 0) <= temp(15 DOWNTO 0);
      temp := to_qsim_state(to_integer('0'&pc1(31 DOWNTO 16),0)+tempint,17);
      pc1(31 DOWNTO 16) <= temp(15 DOWNTO 0);
    END IF;
  END IF;
  END PROCESS do_pc1;
```

```
MX4A_SPEC =
⊢def ∀orep op imm26 pc addr op3 out cnd incpc.
    MX4A_SPEC orep(op,imm26,pc,incpc,addr,cnd,op3,out) =
    (∀t.
      let ctl1 =
            (op3 t) IN {5,6,57} ∨
            (op3 t) IN {54,48,51,50,53,52,49,55} ∧ cnd t  in
       let ctl2 = (op t = 56)  in
        let ctl3 =
              (op3 t) IN {32,34} ∨
              (op t) IN {56,57,5,6} ∨
              (op t) IN {54,48,51,50,53,52,49,55} in
         MUX4_SPEC ctl1 ctl2 ctl3 (addr t) (extend26 orep(imm26 t)) (pc t) incpc (out t))
```

MX2F

Mux MX2F outputs the new value of the pc2 register. If we have to flush the pipeline then the mux outputs the value in the pc4 register. When the pipeline is flushed due to an external interrupt, the value in pc4 is stored in the return register. This is done to restore the state of the processor so that the instructions in the pipeline when the external interrupt takes place will be restarted after the interrupt is serviced. By placing the value in pc2 here, it will get placed into the aluout1 register and then written to the register file in the destination register.

If we are going to wipe[8] the pipeline because of an interrupt, and we have not yet fetched the next instruction then we do not change the value of pc2. Otherwise we pass to pc2 the value that was in pc1.

```
do_pc2 : PROCESS (clk)
  BEGIN
  IF (clk='1' AND clk'LAST_VALUE='0') THEN
    IF (iflush = '1') THEN
        pc2 <= pc4;
    ELSIF (wipe='1' AND op4 /= "000101") THEN
      -- If we're wiping the pipe for an interrupt and we haven't fetched
      -- the next instruction then don't load this register
      NULL; -- Don't change values
    ELSE
      pc2 <= pc1;
    END IF;
  END IF;
  END PROCESS do_pc2;



MX2F_SPEC =
⊢def ∀wipe iflush op pc1 pc4 pc2 out.
    MX2F_SPEC(wipe,iflush,op,pc1,pc4,pc2,out) =
    (∀t.
      let ctl1 = iflush t  in
       let ctl2 = wipe t ∧ ¬(op t) IN {5} in
        MUX3_N_SPEC ctl1 ctl2(pc4 t)(pc2 t)(pc1 t)(out t))
```

[8]The value in the wipe flip-flop and the control line iflush work together to control the pipeline when an external interrupt occurs. For more information about wipe see section 3.6.9.

Mux MX4P

The mux `MX4P` outputs the new value of the `pc4` register. If the instruction in stage four (`op3`) is a branch and it is taken then the address calculated in the ALU is outputted. If `op3` is a Return From Interrupt, a Interrupt, or a Jump On Aligned, then we once again output the value calculated by the ALU. If the instruction in stage four is a JUMP then the value in `pc2` is outputted. If none of these conditions hold then the value of `pc3` is forwarded.

```
do_pc4 : PROCESS (clk)
  -- pc4 is basically keeping track of where to jump in case
  -- of an interrupt
  BEGIN
  IF (clk='1' AND clk'LAST_VALUE='0') THEN
    IF (op3(5 DOWNTO 3) = "110" AND COND = '1') OR
        op3 = "000101" OR op3 = "000110" OR op3 = "111001" THEN
        -- INT, RTI, JALI, or Branch taken
      pc4 <= aluout;
    ELSIF (op3 = "111000") THEN
      -- Jump instruction
      pc4 <= pc2;
    ELSE
      pc4 <= pc3;
    END IF;
  END IF;
  END PROCESS do_pc4;



MX4P_SPEC =
⊢def ∀alu op3 pc2 pc3 out cond.
    MX4P_SPEC(alu,op3,pc2,pc3,cond,out) =
    (∀t.
      let ctl1 = (op3 t) IN {54,48,51,50,53,52,49,55} ∧ cond t
      in
       let ctl2 = (op3 t) IN {5,57,6}
       in
        let ctl3 = (op3 t = 56)
        in
         MUX4_SPEC ctl1 ctl2 ctl3(alu t)(alu t)(pc2 t)(pc3 t)(out t))
```

Mux MX2B

Mux `MX2B` outputs the address of the next word to be fetched from memory. If the instruction executing in the fourth stage is a load or a store then the address forwarded is the memory address calculated by the ALU in the previous stage. If we are not in the fourth stage of a load or store then we forward the value of `pc1` for the next instruction fetch. In the VHDL this mux is described in the process `addressout`.

```
do_addressout : PROCESS (op3,pc1, aluout)
  BEGIN
  IF op3 = "100000" OR op3 = "100010" THEN -- LD or ST instr.
    addressout <= aluout;
  ELSE
    addressout <= pc1;
  END IF;
  END PROCESS do_addressout;



MX2B_SPEC =
⊢def ∀op pc aluout out.
    MX2B_SPEC(op,pc,aluout,out) =
    (∀t.
      let ctl = (op t) IN {32,34}
      in
       MUX_N_SPEC ctl(aluout t)(pc t)(out t))
```

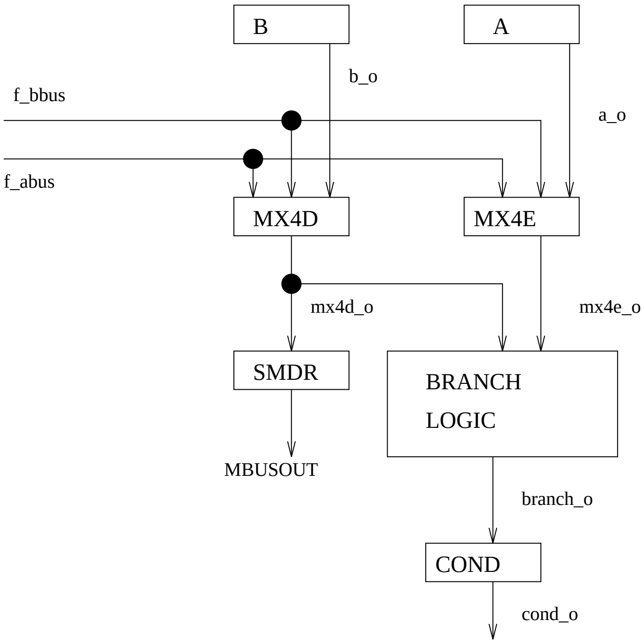

Figure 3.7: The Branch Section of SAWTOOTH

3.6.4 The Branch Section

The branch logic section of SAWTOOTH consists of hardware to compute the new condition code, muxes to forward previous ALU results to the branch logic in case of data hazards, registers holding the values of the two source operands, and the Store Memory Data Register (**smdr**).

Data Forwarding Part II

Just as the ALU had two muxes forwarding in previous results when necessary, the branch logic has two multiplexors that perform the same function. These muxes work in the exact same way as **MX4F** and

MX4G except that they are forwarding the values in the a and b registers. The reason that these muxes not forward the values in the alu1 and alu2 registers is that these two registers are often loaded with either immediate value or the program for calculating effective addresses. The branch logic does not perform such calculations, therefore the values of the a and b registers are forwarded to the branch logic to calculate the condition code used to determine branch targets.

Below are the VHDL process definitions for these muxes and the related HOL specifications.

```
do_mx4d : PROCESS (rb2, rd3, rd4, aluout, aluout1,  b)
  BEGIN
  IF (rb2 = rd3 AND rd3(4 DOWNTO 1) /= "0000") THEN
    mx4d <= aluout;
  ELSIF (rb2 = rd4 AND rd4(4 DOWNTO 1) /= "0000") THEN
    mx4d <= aluout1;
  ELSE
    mx4d <= b;
  END IF;
  END PROCESS do_mx4d;


MX4D_SPEC =
⊢def ∀aluout aluout1 b out rb2 rd3 rd4.
    MX4D_SPEC(b,aluout,aluout1,rb2,rd3,rd4,out) =
    (∀t.
      let ctl1 = (rb2 t = rd3 t) ∧ ¬(rd3 t = R0) in
       let ctl2 = (rb2 t = rd4 t) ∧ ¬(rd4 t = R0) in
        MUX3_N_SPEC ctl1 ctl2(aluout t)(aluout1 t)(b t)(out t))



do_mx4e : PROCESS (ra2, rd3, rd4, aluout, aluout1,  a)
  BEGIN
  IF (ra2 = rd3 AND rd3(4 DOWNTO 1) /= "0000") THEN
    mx4e <= aluout;
  ELSIF (ra2 = rd4 AND rd4(4 DOWNTO 1) /= "0000") THEN
    mx4e <= aluout1;
  ELSE
    mx4e <= a;
  END IF;
  END PROCESS do_mx4e;




MX4E_SPEC =
⊢def ∀aluout aluout1 a out ra2 rd3 rd4 orep.
    MX4E_SPEC orep(a,aluout,aluout1,ra2,rd3,rd4,out) =
    (∀t.
      let ctl1 = (ra2 t = rd3 t) ∧ ¬(rd3 t = R0)  in
       let ctl2 = (ra2 t = rd4 t) ∧ ¬(rd4 t = R0) in
        MUX3_SPEC ctl1 ctl2(aluout t)(aluout1 t)(a t)(out t))
```

The Branch Logic

The Branch Logic is the logic used to determine the new condition code. The new condition code flag is set by analyzing the current instruction and making the appropriate comparison. The condition code is used to determine if a branch is taken. The condition code itself is then stored in a flip-flop.

```
do_cond : PROCESS (clk)
  BEGIN
  IF (clk='1' AND clk'LAST_VALUE='0') THEN
    CASE op2(2 DOWNTO 0) IS
      WHEN "000" => -- BEQ
        IF (mx4d = mx4e) THEN
          cond <= '1';
        ELSE
          cond <= '0';
        END IF;
      WHEN "001" => -- BNEQ
        IF (mx4d = mx4e) THEN
          cond <= '0';
        ELSE
          cond <= '1';
        END IF;
      WHEN "010" => -- BGTZ
        IF (mx4d(31) = '0' AND
           mx4d(30 DOWNTO 0) /= "0000000000000000000000000000000") THEN
          cond <= '1';
        ELSE
          cond <= '0';
        END IF;
      WHEN "011" => -- BGTEZ
        IF (mx4d(31) = '0') THEN
          cond <= '1';
        ELSE
          cond <= '0';
        END IF;
      WHEN "100" => -- BLTZ
        IF (mx4d(31) = '1') THEN
          cond <= '1';
        ELSE
          cond <= '0';
        END IF;
      WHEN "101" => -- BLTEZ
        IF (mx4d(31) = '1' OR
           mx4d(30 DOWNTO 0) = "0000000000000000000000000000000") THEN
          cond <= '1';
        ELSE
          cond <= '0';
        END IF;
      WHEN "110" => -- BC
        cond <= psw(1);
      WHEN "111" => -- BOV
        cond <= psw(0);
      WHEN OTHERS => -- X's and Z's
        cond <= 'X';
    END CASE;
  END IF;
  END PROCESS do_cond;
```

For our **HOL** specification we make extensive use of the abstract functions we defined in our abstract representation.

```
BRANCH_LOGIC_SPEC =
⊢def ∀orep op mx4d mx4e psw cond out.
    BRANCH_LOGIC_SPEC orep(op,mx4d,mx4e,psw,cond,out) =
    (∀t.
      out t =
      ((op t) IN {48} →  eqp orep(mx4e t,mx4d t) |
      ((op t) IN {49} →  ¬eqp orep(mx4e t,mx4d t) |
      ((op t) IN {50} →  gtzp orep(mx4d t) |
      ((op t) IN {51} →  gtezp orep(mx4d t) |
      ((op t) IN {52} →  ltzp orep(mx4d t) |
      ((op t) IN {53} →  ltezp orep(mx4d t) |
      ((op t) IN {54} →  get_cf orep(psw t) |
      ((op t) IN {55} →  get_vf orep(psw t) | cond t))))))))
```

Notice that with the use of higher order logic and the abstract functions the **HOL** specification is much more succinct then the VHDL code. This is almost always the case.

3.6.5 The Destination Registers

In order to keep track of what instruction is executing in each stage, SAWTOOTH keeps multiple copies of the destination register and opcode for each instruction in the pipeline. As shown earlier the mux `MX4C` outputs the destination register as it decodes the instruction in the `ir`. The multiplexor `MX2RD` inputs this value from `MX4C` and forwards either the value of the destination register or zero. The mux plays a very important role in the data forwarding used to feed data into the ALU and branch logic. If the instruction decoded by `MX4C` is a Set Page Address(SPA), Put Program Status Word(PPSW), JUMP, or a branch instruction then we do not need to forward ALU results from previous instructions, so the value of `rd2` will be set to zero. You will notice that in the definitions of the forwarding muxes (`MX4D`, `MX4E`, `MX4F`, and `MX4G`) that each of them made a check to see if the destination register equaled R0 (zero). The destination register `rd2` is also set to zero if the microprocessor is not in supervisory mode (`get_sm orep (psw t)`) and the destination register is a supervisory register (i.e. the page address is set high). The destination register is also set to zero if the current instruction is not a valid one.

In the case that none of these conditions hold, the five bit destination register is extended to six bits with the most significant bit set to the value of the page address bit. The actions of the `MX2RD` mux are described in the process `rd2` in the VHDL.

```
do_rd2 : PROCESS (clk)
  BEGIN
  IF (clk='1' AND clk'LAST_VALUE='0') THEN
    IF (invalid = '1' OR mx4c(31 DOWNTO 26) = "100010" OR
       mx4c(31 DOWNTO 26) = "000001" OR mx4c(31 DOWNTO 26) = "000100" OR
       mx4c(31 DOWNTO 26) = "000110" OR mx4c(31 DOWNTO 26) = "111000" OR
       mx4c(31 DOWNTO 29) = "110") THEN
       -- SPA, PPSW, RTI, JUMP, or BRANCH instructions
      rd2 <= "000000";
    ELSIF (psw(3) = '1' OR pa2 = '0' OR mx4c(25) = '0') THEN
      IF (pa2 = '0') THEN
        rd2(5) <= pa2;
        rd2(4 DOWNTO 0) <= mx4c(25 DOWNTO 21);
      ELSE
        rd2(5) <= mx4c(25);
        rd2(4) <= '0';
        rd2(3 DOWNTO 0) <= mx4c(24 DOWNTO 21);
      END IF;
    ELSE
      rd2 <= "000000";
    END IF;
  END IF;
  END PROCESS do_rd2;


MX2RD_SPEC =
⊢def ∀op pa2 invalid rd orep out psw.
    MX2RD__SPEC orep(op,pa2,rd,psw,invalid,out) =
    (∀t.
      let temp = rd t   in
       let temp2 = extend5 orep(rd t) in
        (out t =
         ((invalid t ∨
           (op t) IN {34,0,1,4,6,56} ∨
           (op t) IN {54,48,51,50,53,52,49,55}) →
          R0 |
          ((get_sm orep(psw t) ∨ ¬pa2 t ∨ ¬bit5 orep 5(rd t)) →
           ((¬pa2 t) →
            setb6 orep(5,extend5 orep(rd t),pa2 t) |
            setb6 orep(5,setb6 orep(4,temp2,F),bit5 orep 5(rd t))) |
           R0))))
```

As the instruction progresses through successive stages in the pipeline, the value kept in **rd2** is stored in and passed on to registers whose specifications are shown below. The value outputted by the mux **MX2RD** is stored in **rd2**. As the instruction passes down the pipeline, its destination register is passed from **rd2** to two other flip-flops: **rd3** and **rd4**. The specifications for **rd3** and **rd4** are a little simpler than that of **rd2**.

For both **rd3** and **rd4**, the VHDL states that if we have an interrupt flush condition then the flip-flop is loaded with a zero. The reason this is done is that register zero is a read-only register and a write-back to that register would be a NOOP.

Since both **rd3** and **rd4** are the same, we generalize their behavior with the **HOL** definition RD_REG6_SPEC. In the data path definition, shown in section 3.6.11, we will create **rd3** and **rd4** with two instantiations of this definition.

```
do_rd3 : PROCESS (clk)
  BEGIN
  IF (clk='1' AND clk'LAST_VALUE='0') THEN
    IF (iflush = '1') THEN
      rd3 <= "000000";
    ELSE
      rd3 <= rd2;
    END IF;
  END IF;
  END PROCESS do_rd3;

do_rd4 : PROCESS (clk)
  BEGIN
  IF (clk='1' AND clk'LAST_VALUE='0') TH    IF (iflush = '1') THEN
      rd4 <= "000000";
    ELSE
      rd4 <= rd3;
    END IF;
  END IF;
  END PROCESS do_rd4;



RD_REG6_SPEC =
⊢def ∀r clk iflush out.
    RD_REG6_SPEC(r,clk,iflush,out) =
    (∀t. out(t + 1) = (clk t →  (iflush t →  R0 | r t) | out t))
```

3.6.6 The Opcode Registers

Just as the destination register for an instruction was saved as it went from one stage to another, the opcode of the instruction is also kept in successive registers. For example, once the instruction is decoded its opcode is stored in the register `op2`. This value is used to control the execution of the third stage of the pipeline. As the instruction moves to the fourth stage, the instruction's opcode is passed to `op3`. And finally once the instruction finishes executing in stage four the value is passed to `op4`. We model the behavior of these registers with the following **HOL** definitions. As we did with the destination registers, we use one general definition to model the behavior of `op3` and `op4`.

```
do_op2 : PROCESS (clk)
  BEGIN
  IF (clk='1' AND clk'LAST_VALUE='0') THEN
    op2 <= mx4c(31 DOWNTO 26);
  END IF;
  END PROCESS do_op2;

do_op3 : PROCESS (clk)
  BEGIN
  IF (clk='1' AND clk'LAST_VALUE='0') THEN
    IF(iflush = '1') THEN
      op3 <= "000000";
    ELSE
      op3 <= op2;
    END IF;
  END IF;
  END PROCESS do_op3;

do_op4 : PROCESS (clk)
  BEGIN
  IF (clk='1' AND clk'LAST_VALUE='0') THEN
    IF(iflush = '1') THEN
      op4 <= "000000";
    ELSE
      op4 <= op3;
    END IF;
  END IF;
  END PROCESS do_op4;



OP2_REG6_SPEC =
⊢def ∀inst ld op2.
    OP2_REG6_SPEC(inst,ld,op2) =
    (∀t. op2(t + 1) = (ld t →  inst t | op2 t))



OP_REG6_SPEC =
⊢def ∀op clk iflush out.
    OP_REG6_SPEC(op,clk,iflush,out) =
    (∀t. out(t + 1) = (clk t →  (iflush t →  0 | op t) | out t))
```

Notice that the definition for `op2` is slightly simpler then the other opcode flip-flop definition. This is because for the later opcode flip-flops (`op3` and `op4`) if we are going to flush the pipeline, then the value zero is inserted representing a NOOP instruction.

3.6.7 The Program Status Word

The mux `MX4PSW` outputs the new value of the program status word after every clock cycle. The mux receives four inputs: the control line `reset`, the control line `iflush`, `npsw` (calculated by the ALU), and the previous program status word `pswo`. If the microprocessor is to be reset then the program status word reset to all zeros except the interrupt enable flag which is set high.

The program word is actually only 8 bits wide, but we keep it in a 32 bit word allowing us to keep multiple copies without having extra hardware. Whenever an interrupt occurs the program status word is shifted left 8 bits, thus giving us a copy of the old program status word. When the RTI instruction is executed, the program status word is shifted right to restore the old **psw**.

If the pipeline is to be flushed the copy of the previous program status word is once again made the **psw**. If we have performed an ADD instruction, a SUB instruction , a shift operation, a SPA instruction, a PPSW instruction, an INT instruction, or an RTI instruction the mux outputs the value of **npsw** which hold the value of the updated program status word calculated by the ALU.

```
do_psw : PROCESS (clk)
  BEGIN
  IF (clk='1' AND clk'LAST_VALUE='0') THEN
    IF (reset='1') THEN
      psw <= "00000100000000100000000100000000100";
    ELSIF(iflush = '1') THEN
      psw <= pswo1;
    ELSIF (op2(5 DOWNTO 3) = "001" OR op2(5 DOWNTO 3) = "010") AND
          (op2(2) = '0') THEN -- We're adding or subtracting
      psw <= npsw; -- npsw comes from ALU
    ELSIF (op2(5 DOWNTO 3) = "011" AND (op2(2 DOWNTO 0) = "000" OR
           op2(2 DOWNTO 0) = "001" OR op2(2 DOWNTO 0) = "010")) THEN
           -- We're shifting right or left
      psw <= npsw; -- npsw comes from ALU
    ELSIF op2(5 DOWNTO 3) = "000" AND
          (op2(2 DOWNTO 0) = "001" OR op2(2 DOWNTO 0) = "100" OR
          op2(2 DOWNTO 0) = "101" OR op2(2 DOWNTO 0) = "110") THEN
          -- We're executing SPA, PPSW, INT, or RTI
      psw <= npsw; -- npsw comes from ALU
    ELSE
      NULL; -- Don't change psw
    END IF;
  END IF;
  END PROCESS do_psw;


MX4PSW_SPEC =
⊢def ∀orep reset iflush op npsw psw pswo out.
    MX4PSW_SPEC orep(reset,iflush,op,npsw,psw,pswo,out) =
    (∀t.
      let ctl1 = reset t  in
       let ctl2 = iflush t  in
        let ctl3 =
              (op t) IN {8,9,10,11,16,17,18,19} ∨
              (op t) IN {24,25,26} ∨
              (op t) IN {1,4,5,6}
        in
         let a = ((repeat orep(32,F)) except [ie,T;10,T;18,T;26,T])orep in
          let b = pswo t  in
           let c = npsw t  in
            let d = psw t in
          MUX4_N_SPEC ctl1 ctl2 ctl3 a b c d(out t))
```

3.6.8 The Page Address: Pa2

As mentioned earlier SAWTOOTH has three pages of 16 registers–two of which are visible to the programmer at any one time. Each instruction format specifies a five bit address for a registers. Before the register contents can be fetched from or written to the register file, the page address bit is used to extend the five bit register address to a six bit address. The mux **MX4PA** updates the value of the **pa2** flip-flop each clock cycle.

The mux receives six inputs: the control line `reset`, the control line `iflush`, the opcode from `MX4C`, the 16 bit immediate from `MX4C`, the program status word, and the current value of the page address.

If the microprocessor is to be reset then the page address is set low. If the pipeline is to be flushed then the page address bit is set to the value held in the program status word (`get_pa orep (psw t)`). If the instruction just decoded is a SPA (set page address), then the page address is set to the lowest bit in the 26 bit immediate. If none of these conditions hold the value of the page address remains unchanged.

```
do_pa2 : PROCESS (clk)
  BEGIN
  IF (clk='1' AND clk'LAST_VALUE='0') THEN
    IF (reset='1') THEN
      pa2<='0';
    ELSIF(iflush = '1') THEN -- We received an interrupt so restore state
      pa2<=pswo1(4);
    ELSIF (mx4c(31 DOWNTO 26) = "000001") THEN
      pa2<=mx4c(0);
    ELSE -- Don't load pa2
      NULL;
    END IF;
  END IF;
  END PROCESS do_pa2;



MX4PA_SPEC =
⊢def ∀orep reset iflush op imm pa2 psw out.
    MX4PA_SPEC orep(reset,iflush,op,imm,psw,pa2,out) =
    (∀t.
      let ctl1 = reset t
      and ctl2 = iflush t
      and ctl3 = (op t = 1) in
       let b = get_pa orep(psw t) in
        let c = bit orep 0(extend orep(imm t)) in
         let d = pa2 t in
           MUX4_1_SPEC ctl1 ctl2 ctl3 F b c d(out t))
```

3.6.9 The Wipe Flip-flop

The value kept in the `wipe` flip-flop is used to help control the actions of the pipeline when an external interrupt occurs.

The mux `MX4WIPE` outputs the new value of the wipe flip-flop. The mux receives four inputs: the control line `reset`, the control line `iflush`, the value of the `op4` flip-flop, and the current value of `wipe`. If the microprocessor is to be reset then `wipe` is set low.

If the pipeline is to be flushed then wipe is set to high. The `iflush`[9] line signals that an external interrupt has occurred. It should be noted that here that the `iflush` can only be high if the `wipe` flip-flop is low. After `wipe` is set high, stays high until the interrupt instruction calling the appropriate service routine finishes. This prevents the `iflush` line from being set high again until the INT instruction is in stage five. Keeping `wipe` high also prevents instructions from entering the pipeline until the INT

[9]The specification for the `iflush` control signal is presented in section 3.7.

instruction jumps the service routine sine the `MX4C` mux places NOOPS into the pipeline if `wipe` is high. When the INT instruction for an external interrupt enters stage five, `wipe` is set low.

If none of these conditions hold then the value of `wipe` remains unchanged.

```
do_wipe : PROCESS (clk)
  BEGIN
  IF (clk='1' AND clk'LAST_VALUE='0') THEN
    IF (reset = '1') THEN
      wipe <= '0';
    ELSIF (iflush = '1') THEN
      wipe <= '1';
    ELSIF op4 = "000101" THEN -- op4 = INT instruction
      wipe <= '0';
    ELSE
      NULL;
    END IF;
  END IF;
  END PROCESS do_wipe;


MX4WIPE_SPEC =
⊢def ∀reset iflush wipe op out.
    MX4WIPE_SPEC(reset,iflush,op,wipe,out) =
    (∀t.
      let ctl1 = reset t in
       let ctl2 = iflush t in
        let ctl3 = (op t) IN {5}
        in
         MUX4_1_SPEC ctl1 ctl2 ctl3 F T F(wipe t)(out t))
```

3.6.10 External Memory Specification

Traditionally, in microprocessor verification, memory has been considered part of the state of the microprocessor. For the SAWTOOTH specification we follow this tradition.

As we did for bit strings, we define the operations that operate on memory with an abstract representation (see section 3.2). We define two functions over the uninterpreted type `*memory`: `fetch` and `store`.

In the definition below memory is only updated if the `wr_s` is set high. This input is a control line which designates whether or not a write is to be performed. If we are not writing to memory then the value of memory goes unchanged and the value going out on the `data2` line is set to the value fetched from the address coming in from the `addr` line. The address coming in here is the address outputted from the mux `MX2B` and is either the address of the next instruction to fetch or is the address for a LOAD or STORE.

In this definition, `data1` refers to the line `MBUSOUT` in Figure 3.1, and `data2` refers to the line `MBUSIN`. This definition is somewhat misleading. Looking at this definition, one would get the impression that there are two buses to memory. However we already know that there is only one bus to memory because we must stall an instruction fetch is we are doing a LOAD or STORE. What is really happening is that two separate memory lines exit the microprocessor (`MBUSIN` and `MBUSOUT`). Outside the processor, the two lines are connected to the single memory bus via a bidirectional pad which controls which line has access to the bus.

```
MEM_SPEC =
⊢def ∀mrep orep wr_s addr data1 data2 mem clk.
    MEM_SPEC orep mrep wr_s addr data1 data2 mem clk =
    (∀t.
      (mem(t + 1) =
       (clk t →
        (wr_s t →
         store mrep(mem t,address orep(addr t),data1 t) |
         mem t) |
        mem t)) ∧
      (¬wr_s t ⟹  (data2 t = fetch mrep(mem t,address orep(addr t)))))
```

3.6.11 The Data Path Definition

The definition of the Data Path is simply a conjunction of the block definitions presented so far. As done with traditional hardware verification in **HOL** [CGM87] and similar microprocessor verification efforts [Win90b, Joy89], existential quantification is used to represent hidden internal lines. The definition for the Data Path is shown in Figure 3.8.

3.7 Control Block

Since SAWTOOTH is not a microprogrammed machine, the control block is a much simpler piece of hardware in comparison to the control blocks of other verified microprocessors [Win90b].

Here the only function of the control block is to calculate the values for the `bflush`, `iflush`, `invalid`, and `wr` control lines. The `bflush` signifies that a branch will be taken. The `iflush` line is set to high if there is an exception and we are not is supervisory mode and we are not wiping the pipeline. The `invalid` flag is set to high if the opcode of the instruction decoded by `MX4C` is not valid. The `wr` line is set to high if the instruction executing in the fourth stage is a LOAD or a STORE. The `wr` line is used coordinate reads and writes to memory.

```
DATA_PATH =
⊢def ∀orep decrep mrep reg psw pc1 pc2 pc3 mem pc4 alu1 alu2 aluout
   aluout1 a b ir smdr pswo cond ra2 rb2 rd2 rd3 rd4 op2 op3 op4 pa2
   wipe iflush bflush invalid wr exc reset clk ivec clk_1 clk_2.
   DATA_PATH orep decrep mrep reg psw pc1 pc2 pc3 pc4 mem alu1
   alu2 aluout aluout1 a b ir smdr pswo op2 op3 op4 ra2 rb2 rd2
   rd3 rd4 cond pa2 wipe iflush bflush invalid exc reset ivec
   clk_1 clk_2 wr =
   (∀t.
     ∃ir_o mxir_o mx4c_o1 mx4c_o2 mx4c_o3 mx4c_o4 mx4c_o5 mx4c_o6
     pc1_o pc2_o pc3_o pc4_o fetch0 fetch1 stackout1 stackout0 f_abus
     f_bbus b_o a_o mx4d_o mx4e_o smdr_o branch_o pswo_o mx2f_o
     addressout muxout1 muxout2 muxout3 alu_o alu_src1 alu_src2
     mbusin mbusout mx4psw_o npsw mx4wipe_o mx2d_o carry_o mx4b_o
     ncond_o mx4pa_o muxout_pc mx2rd_o psw_o mx4p_o.
     REG_SPEC mx2d_o clk_2 alu_src1 alu1 ∧
     REG_SPEC mx4b_o clk_2 alu_src2 alu2 ∧
     MX4F_SPEC orep(alu_src1,f_abus,f_bbus,op2,ra2,rd3,rd4,muxout1) ∧
     MX4G_SPEC orep(alu_src2,f_abus,f_bbus,op2,rb2,rd3,rd4,muxout2) ∧
     ALU_SPEC orep (clk_2,muxout1,muxout2,psw,op2,carry_o,f_abus,alu_o,npsw) ∧
     CARRY_LOGIC_SPEC orep(op2,psw,carry_o) ∧
     REG_SPEC alu_o clk_2 f_abus aluout ∧
     MX4H_SPEC(mbusin,f_abus,op3,pc2_o,muxout3) ∧
     REG_SPEC muxout3 clk_2 f_bbus aluout1 ∧
     REG_SPEC muxout_pc clk_2 pc1_o pc1 ∧
     MX2F_SPEC(wipe,iflush,op4,pc1_o,pc4_o,pc2,mx2f_o) ∧
     REG_SPEC mx2f_o clk_2 pc2_o pc2 ∧
     REG_SPEC pc2_o clk_2 pc3_o pc3 ∧
     MX4P_SPEC(f_abus,op3,pc2_o,pc3_o,cond,mx4p_o) ∧
     REG_SPEC mx4p_o clk_2 pc4_o pc4 ∧
     (MX4A_SPEC orep (mx4c_o1,mx4c_o6,pc1_o,inc orep(pc1_o t),f_abus,cond,op3,muxout_pc) ∧
      MX2B_SPEC(op3,pc1_o,f_abus,addressout)) ∧
     MXIR_SPEC orep(bflush,op3,mx4c_o1,pc1_o,mbusin,mxir_o) ∧
     REG_SPEC mxir_o clk_2 ir_o ir ∧
     MX4C_SPEC orep decrep (wipe,invalid,bflush,iflush,ivec,ir_o,mx4c_o1,mx4c_o2,mx4c_o3,mx4c_o4,mx4c_o5,mx4c_o6) ∧
     MX2RD__SPEC orep(mx4c_o1,pa2,mx4c_o2,psw,invalid,mx2rd_o) ∧
     RD_REG6_SPEC(mx2rd_o,clk_2,iflush,rd2) ∧
     RD_REG6_SPEC(rd2,clk_2,iflush,rd3) ∧
     RD_REG6_SPEC(rd3,clk_2,iflush,rd4) ∧
     OP2_REG6_SPEC(mx4c_o1,clk_2,op2) ∧
     OP_REG6_SPEC(op2,clk_2,iflush,op3) ∧
     OP_REG6_SPEC(op3,clk_2,iflush,op4) ∧
     MX4PA_SPEC orep(reset,iflush,mx4c_o1,mx4c_o5,pswo,pa2,mx4pa_o) ∧
     FF_SPEC mx4pa_o clk_2 pa2 ∧
     FETCH1_SPEC orep(mx4c_o1,mx4c_o2,mx4c_o4,pa2,fetch1) ∧
     FETCH0_SPEC orep(pa2,mx4c_o3,fetch0) ∧
     RA_REG6_SPEC(clk_2,fetch0,ra2) ∧
     RA_REG6_SPEC(clk_2,fetch1,rb2) ∧
     REGISTER_BLOCK orep (clk_1,clk_2,fetch0,fetch1,f_bbus,rd4,stackout0,stackout1,reg) ∧
     MX2D_SPEC(mx4c_o1,pc1_o,stackout0,mx2d_o) ∧
     MX4B_SPEC orep(mx4c_o1,mx4c_o5,stackout1,mx4b_o) ∧
     MX4PSW_SPEC orep(reset,iflush,op2,npsw,psw,pswo,mx4psw_o) ∧
     REG_SPEC mx4psw_o clk_2 psw_o psw ∧
     REG_SPEC psw_o clk_2 pswo_o pswo ∧
     REG_SPEC stackout0 clk_2 a_o a ∧
     REG_SPEC stackout1 clk_2 b_o b ∧
     MX4D_SPEC(b_o,f_abus,f_bbus,rb2,rd3,rd4,mx4d_o) ∧
     MX4E_SPEC orep(a_o,f_abus,f_bbus,ra2,rd3,rd4,mx4e_o) ∧
     BRANCH_LOGIC_SPEC orep(op2,mx4d_o,mx4e_o,psw,cond,ncond_o) ∧
     REG_SPEC mx4d_o clk_2 mbusout smdr ∧
     FF_SPEC ncond_o clk_2 cond ∧
     MX4WIPE_SPEC(reset,iflush,op4,wipe,mx4wipe_o) ∧
     FF_SPEC mx4wipe_o clk_2 wipe ∧
     MEM_SPEC orep mrep wr addressout mbusout mbusin mem clk_2)
```

Figure 3.8: The Datapath Definition for SAWTOOTH.

```
CONTROL_BLOCK =
⊢def ∀orep exc decrep wipe psw out1 out2 out3 op3 cond ir out4.
    CONTROL_BLOCK
    orep
    decrep
    (exc,wipe,psw,op3,cond,ir,out1,out2,out3,out4) =
    (∀t.
      let ifl = exc t ∧ ¬bit orep sm(psw t) ∧ ¬wipe t   in
       let inst =
             ((ir t = wordn orep 0) →   INST_Decode NOOP | Wordn_decode decrep(ir t))  in
        let op = FST inst  in
         ((out1 t = ifl) ∧
          (out2 t = ((op3 t) IN {54,48,51,50,53,52,49,55} ∧ cond t ∨ (op3 t) IN {57,5,6}) ∧ ¬ifl) ∧
          (out3 t =  ¬op IN
            {32,33,34,8,9,10,11,12,13,14,16,17,18,19,20,21,22,23,24,25,
             26,27,28,29,1,2,3,4,5,6,0,56,57,48,49,50,51,52,53,54,55}) ∧
          (out4 t = (op3 t) IN {34})))
```

3.8 The Clock

SAWTOOTH uses a two phase clock–that is for every clock cycle there is two phases. We define the clock to be a two valued counter alternating between **TRUE** and **FALSE**. The clock is in phase one if **clk_1** is high and in phase two is **clk_2** is high.

```
CLOCK_SPEC =
⊢def ∀clk clk_1 clk_2.
    CLOCK_SPEC clk clk_1 clk_2 =
    (∀t.
      (clk(t + 1) = ¬clk t) ∧ (clk_1 t = clk t) ∧ (clk_2 t = ¬clk t))
```

3.9 The EBM Definition

In this section we present the final EBM definition for the SAWTOOTH microprocessor.

Using the state and environment data types we defined earlier, we now define electronic block model specification.

```
sawtoothEBM =
⊢def ∀orep mrep decrep s e p.
    sawtoothEBM orep mrep decrep s e p =
    (∃iflush bflush invalid wr clk clk_1 clk_2.
      CONTROL_BLOCK
      orep
      decrep
      (Excp o e,Wipee o s,Pswe o s,Op3e o s,Conde o s,Ire o s,iflush,
       bflush,invalid,wr) ∧
      CLOCK_SPEC clk clk_1 clk_2
      ∧
      DATA_PATH orep decrep mrep (Rege o s) (Pswe o s) (Pc1e o s)
      (Pc2e o s) (Pc3e o s) (Pc4e o s) (Meme o s) (Alu1e o s)
      (Alu2e o s) (Aluoute o s) (Aluout1e o s) (Ae o s) (Be o s)
      (Ire o s) (Smdre o s) (Pswoe o s) (Op2e o s) (Op3e o s)
      (Op4e o s) (Ra2e o s) (Rb2e o s) (Rd2e o s) (Rd3e o s)
      (Rd4e o s) (Conde o s) (Pa2e o s) (Wipee o s)  iflush
      bflush  invalid  (Excp o e)  (Resetp o e)  (Ivecp o e)
      clk_1  clk_2   wr)
```

The EBM is defined to be the conjoining of the Data Path and the Control Block and the Clock Specification. Once again existential quantification is used to reflect hidden lines. The `o` notation used between the state selector functions and the state variable `s` is used to denote function composition. In this definition we wish to make the EBM look like an interpreter so the state is a variable of the type `:time->EBM_state`. However our state selector functions are defined to be of the type `:EBM_state ->*` where the $*$ represents a state element type. In order to get the values of state elements from `s` we compose the two functions. For example consider the expression: `Pc2e o s`.

- `s` is of the type `:time->EBM_state`.
- `Pc2e` is of the type `:EBM_state -> *wordn`.
- When we compose the two functions with the expression `(Pc2e o s)` we get a new function of the type `:time->*wordn` which is the type of arguments expected by the Control Block and Data Path definitions.

3.10 Summary

In this Chapter we have presented the EBM specification of the SAWTOOTH microprocessor. This specification of the EBM is a *structural* description of the processor. This EBM specification for SAWTOOTH was written by hand translation of the designers behavioral VHDL. Therefore the specifications presented here maybe at a slightly higher level of abstraction then other EBM specifications in the past [Joy89, Win90b].

Chapter 4

The Phase and Pipeline Levels

In this chapter we present the specifications of the phase level and pipeline level of the SAWTOOTH microprocessor. The two levels are very closely related. The phase level is a behavioral description of the electronic block model. Each phase level instruction describes the actions that take place during one phase of a clock cycle. At the pipeline level, we model the execution of the pipeline as one large instruction since the five stages all execute concurrently. This pipeline instruction is constructed from the two phase level instructions. We begin the specifications with the phase level.

4.1 Pipelining Conventions

The SAWTOOTH microprocessor uses instruction pipelining which concurrently executes 5 instructions. To avoid confusion, we use the following conventions when referencing instructions in the pipeline:

- `Op2` is the instruction *executing* in the third stage of the pipeline.[1]
- `Op3` is the instruction executing in the fourth stage of the pipeline.
- `Op4` is the instruction executing in the fifth stage of the pipeline.

4.2 The Phase Level Specification

In this section we present the specification of the phase level. The specification employs the interpreter model developed and subsequently revised by Windley [Win90a, Win93]. The generic interpreter theory is an abstract theory built with Windley's abstract theory package for **HOL** [Win92]. In our specification we define a set of concrete functions used to instantiate the abstract functions of the underlying theory. For the generic interpreter theory, the set of functions we will specify is shown in Table 4.1.

[1] The reason that the op2 is used to reference the instruction in the third stage is that the state element value for `op2` is calculated in the second stage–thus `op2`.

Table 4.1: The abstract functions and their types for the interpreter model.

Operation	*Type*
`instructions`	`":*key -> (*state->*env->*state)"`
`select`	`":*state->*env->*key"`
`output`	`":*key->(*state->*env->*out)'"`
`subout`	`":*out'->*out"`
`substate`	`":*state'->*state"`
`subenv`	`":*env'->*env"`
`Impl`	`":(time'->*state')->(time'->*env')->(time'->*out') ->bool"`
`sync`	`":*state'->*env'->bool"`

4.2.1 The Phase State

The state of the microprocessor at the phase level is the same as the state at the EBM level presented in section 3.3.

4.2.2 The Phase Environment

The environment of SAWTOOTH is the same at all levels of our specification and was given in section 3.3.1.

4.2.3 The `instructions` Function

The GIT is based on the idea that an interpreter takes an input of a state and environment, executes a function on the state called a state transition function, and returns a new state. In the abstract theory the function that performs this action is the `instructions` function. The `instructions` functions takes as input a key and returns the appropriate state transition function relating to that key.

For the phase level of SAWTOOTH, there are two phase instructions: `PHASE1_SPEC` and `PHASE2_SPEC`.

4.2.4 Phase 1

`PHASE1_SPEC` specifies the actions executed in the first phase of a clock cycle. In this phase the register file write back is executed and the clock is updated.

```
PHASE1_SPEC=
⊢def ∀decrep orep mrep s e.
    PHASE1_SPEC decrep orep mrep s e =
    (let reg = Rege s
     and psw = Pswe s
     and pc1 = Pc1e s
     and mem = Meme s
     and pc2 = Pc2e s
     and pc3 = Pc3e s            ----state selector functions
     and pc4 = Pc4e s
     and alu1 = Alu1e s
     and alu2 = Alu2e s
     and aluout = Aluoute s
     and aluout1 = Aluout1e s
     and a = Ae s
     and b = Be s
     and ir = Ire s
     and smdr = Smdre s
     and pswo = Pswoe s
     and cond = Conde s
     and ra2 = Ra2e s
     and rb2 = Rb2e s
     and op2 = Op2e s
     and op3 = Op3e s
     and op4 = Op4e s
     and rd2 = Rd2e s
     and rd3 = Rd3e s
     and rd4 = Rd4e s
     and wipe = Wipee s
     and clk = Clke s
     and pa2 = Pa2e s
     and exc = Excp e
     and ivec = Ivecp e
     and reset = Resetp e
     in
      let new_clk = ¬clk   in
       let new_reg = UPDATE_REG orep rd4 reg aluout1   in
        EBM_State new_reg psw pc1 mem pc2 pc3 pc4 alu1 alu2
        aluout aluout1 a b ir smdr pswo cond ra2 rb2 op2 op3
        op4 rd2 rd3 rd4 pa2 wipe new_clk)
```

In `PHASE2_SPEC` the rest of the necessary state transitions take place. This definition is very large, so it is presented in parts by what is done in each stage of the pipeline, but it is really just one function.

4.2.5 Phase2: Stage 1

In the first stage of the pipeline we perform three basic tasks: we update the program counter, we update the instruction register, and we update control lines (`iflush, bflush, and invalid`).

```
PHASE2_SPEC =
⊢def ∀decrep orep mrep s e.
   PHASE2_SPEC decrep orep mrep s e =

   ....selector functions deleted....

  let iflush = exc ∧ ¬get_sm orep psw ∧ ¬wipe  in          ---update control lines
  let bflush =
         (op3 IN {54,48,51,50,53,52,49,55} ∧ cond ∨
          op3 IN {57,5,6}) ∧  ¬iflush  in
  let new_clk = ¬clk in
  let mx4c_temp =  ((ir = wordn orep 0) →   INST_Decode NOOP |
                                  Wordn_decode decrep ir)  in
   let op_temp = FST mx4c_temp  in
   let invalid =  ¬op_temp IN
              {32,33,34,8,9,10,11,12,13,14,16,17,18,19,20,21,22,23,
               24,25,26,27,28,29,1,2,3,4,5,6,0,56,57,48,49,50,51,52,
               53,54,55}  in
    let mx4c = (iflush →    INST_Decode(INT R1 R0(low16w orep ivec)) |
               ((wipe ∨ invalid ∨ bflush) →   INST_Decode NOOP |
                mx4c_temp)) in
    let op = FST mx4c in                                   ---update ir
    let irtemp =  ((op3 IN {32,34} ∨ bflush) →    wordn orep 0 |
               (op IN {5,6,56,57,54,48,51,50,53,52,49,55} →  wordn orep 0 |
                  fetch mrep(mem,address orep pc1)))  in
    let new_ir = irtemp
       and new_pc1_temp = inc orep pc1  in
    let new_wipe =  (reset →  F | (iflush →  T | (op4 IN {5} →  F | wipe)))  in
```

The multiplexor **mx4c**, which does the instruction decoding, is assigned a value in the first stage when it is actually a second stage action. The reason for this is the use of the functional language for the specification. The opcode residing in the **mx4c** mux is needed to compute the new value for the instruction register. Because of the semantics of the let notation the assignment of a value for **mx4c** must appear before the let statement for **new_ir**. In reality these two actions are occurring in parallel in hardware, but it is difficult to model this parallel behavior in the ML language properly.

4.2.6 Phase2: Stage 2

In stage two of the pipeline, the instruction in the **mx4c** multiplexor is decoded. The destination register **rd2**, opcode register **op2**, the page address flip-flop **pa2**, the source 1 register **ra2**, and the source 2 register **rb2** are all loaded with values determined from the decoded instruction. The values for the ALU source registers, **alu1** and **alu2**, and the branch source registers, **a** and **b**, are fetched from the register file.

There are a couple of important things to notice about this stage. The first of which is the multiplexor **mx2rd**. This mux outputs the value of the destination register. If the instruction being decoded does not perform a register write back then the destination register is set to **R0**. Register **R0** is a read only register that contains the value zero. A write to register **R0** does nothing. If the instruction does perform a register write back the mux outputs the destination register from the instruction with the page address bit properly appended to it.

```
let rd = FST(SND mx4c)                              --decode the instruction
and ra = FST(SND(SND mx4c))
and rb = FST(SND(SND(SND mx4c)))
and imm' = FST(SND(SND(SND(SND mx4c))))
and imm26 = SND(SND(SND(SND(SND mx4c)))) in                --calculate new dest. register
let mx2rd =   (op IN {34,1,4,6,56,54,48,51,50,53,52,49,55} →  R0 |
              (invalid →   R0 |
              ((get_sm orep psw ∨ ¬pa2 ∨ ¬bit5 orep 5 rd) →
                     ((¬pa2) →  setb6 orep(5,extend5 orep rd,pa2) |
                      setb6 orep (5,setb6 orep(4,extend5 orep rd,F),
                       bit5 orep 5 rd)) |
                     R0)))  in
 let imm = extend orep imm'   in                                --calculate new program counter
 let mx4a = (op3 IN {5,6,57} →  aluout |
           ((op3 IN {54,48,51,50,53,52,49,55} ∧ cond) →  aluout |
           (op IN {56} →  extend26 orep imm26 |
            ((op3 IN {32,34} ∨  op IN {56,57,5,6} ∨
             op IN {54,48,51,50,53,52,49,55}) →    pc1 | new_pc1_temp)))) in
  let new_pc1 = mx4a
      and mx2f = (iflush →    pc4 | ((wipe ∧ ¬op4 IN {5}) →  pc2 | pc1)) in
  let new_pa2 = (reset →  F |
              (iflush →  get_pa orep pswo |
              ((op = 1) →  bit orep 0 (extend26 orep imm26) | pa2)))  in
  let fetch0 = ((¬pa2) →  setb6 orep(5,extend5 orep ra,pa2) |           --calculate 6 bit register addresses
                       setb6 orep
                       (5,setb6 orep(4,extend5 orep ra,F),
                        bit5 orep 5 ra))   in
  let fetch1 = ((op IN {54,48,51,50,53,52,49,55} ∨ op IN {34}) →
                   ((¬pa2) →  setb6 orep(5,extend5 orep rd,pa2) |
                         setb6 orep
                         (5,setb6 orep(4,extend5 orep rd,F),
                          bit5 orep 5 rd)) |
                    ((¬pa2) →  setb6 orep(5,extend5 orep rb,pa2) |
                         setb6 orep
                         (5,setb6 orep(4,extend5 orep rb,F),
                          bit5 orep 5 rb))) in
  let mx2d = (op IN {54,48,51,50,53,52,49,55} →  pc1 | reg fetch0)    ---calculate alu1
  in
  let new_alu1 = mx2d
  in                                                  ---calculate alu2
  let mx4b = (op IN {8,9,10,11,12,13,14,1,2,3,4,5,6} →  imm |
            (op IN {32,34,33} →  signextend orep imm' |
             ((op IN {54,48,51,50,53,52,49,55} ∨ op IN {56,57}) →
                               signextend orep imm' |
                               reg fetch1)))   in
  let new_alu2 = mx4b  in
  let new_pc2 = mx2f
     and new_op2 = op
     and new_rd2 = mx2rd
     and new_a = reg fetch0
     and new_b = reg fetch1
     and new_ra2 = fetch0
     and new_rb2 = fetch1  in
```

Phase2: Stage 3

The **HOL** specification of the stage three actions is shown below.

```
let mx4f = (op2 IN {54,48,51,50,53,52,49,55} →   alu1 |
          (Data_hazard orep ra2 rd3 →   aluout |
          (Data_hazard orep ra2 rd4 →   aluout1 |  alu1)))  in                                          ––––data forwarding
let mx4g = ((¬op2 IN {16,17,18,19,20,21,22,23}) →  alu2 |
          (Data_hazard orep rb2 rd3 →  aluout |
          (Data_hazard orep rb2 rd4 →  aluout1 | alu2)))  in
 let carry =  (op2 IN {16,8} →   F |                                  –––carry flag
             (op2 IN {17,9} →  get_cf orep psw |
             (op2 IN {18,10} →   T |
             (op2 IN {19,11} →  get_cf orep psw | F))))  in
 let new_aluout =  ALU_func orep op2 psw carry (mx4f,mx4g) in
 let new_pc3 = pc2   in
 let new_psw =
      (reset →   ((repeat orep(32,F)) except [ie,T;10,T;18,T;26,T]) orep |
      (iflush →   pswo |
      PSW_func orep op2 psw carry (mx4f,mx4g))) in
 let new_pswo = psw in                                        –––branch data forwarding
 let mx4e = (Data_hazard orep ra2 rd3 →  aluout |
           (Data_hazard orep ra2 rd4 →  aluout1 | a))  in
 let mx4d = (Data_hazard orep rb2 rd3 →  aluout |
          (Data_hazard orep rb2 rd4 →  aluout1 | b))  in
 let new_smdr = mx4d  in                                       ––cond code
 let new_cond = (op2 IN {54} →  bit orep cf psw |
                  (op2 IN {55} →  bit orep vf psw |
                  (op2 IN {48} →  eqp orep(mx4e,mx4d) |
                  (op2 IN {51} →  gtezp orep mx4d |
                  (op2 IN {50} →  gtzp orep mx4d |
                  (op2 IN {53} →  ltezp orep mx4d |
                  (op2 IN {52} →  ltzp orep mx4d |
                  (op2 IN {49} →  ¬eqp orep (mx4e,mx4d) | cond))))))))
      in
  let new_op3 =  (iflush →  0 | op2)
   and new_rd3 =  (iflush →  R0 | rd2) in
```

Stage three of the **PHASE2_SPEC** definition is where most of the processing takes place. In stage three, the ALU computes the new **aluout**, the program status word is updated, the new condition code is computed, and several registers are loaded with values to maintain state. The forwarding multiplexors are once again called **mx4d**, **mx4e**, **mx4f**, and **mx4g**. For the phase level specification of these muxes we define the predicate **Data_hazard** shown below. The ALU is once again modeled with the definition **ALU_func** which was defined in section 3.6.2. Modeling the ALU with this function makes the verification much easier because it hides the details of the ALU until they are necessary.

```
⊢def ∀orep src dest.
    Data_hazard orep src dest = (src = dest) ∧ ¬(dest = R0)
```

Phase2: Stage 4

The main action that occurs in stage four is the memory fetch or store for load and store instructions. If **op3** is a **STORE** instruction then memory is updated with the value in the **smdr** at the address computed by the ALU in stage three. If **op3** is a load instruction then the memory contents at the location computed by the ALU are loaded into the **aluout1** register.

```
let new_mem = (op3 IN {34} → store mrep (mem, address orep aluout, smdr) | mem) in
let mx4h = (op3 IN {32} → fetch mrep (mem, address orep aluout) |
          (op3 IN {5,57} → pc2 | aluout)) in
let new_aluout1 = mx4h
in
let new_pc4 = ((op3 IN {54,48,51,50,53,52, 49,55} ∧ cond) → aluout |
             (op3 IN {5,57,6} → aluout |
             (op3 IN {56} → pc2 | pc3))) in
 let new_op4 = (iflush → 0 | op3)
 and new_rd4 = (iflush → R0 | rd3) in
 EBM_State reg new_psw new_pc1 new_mem new_pc2 new_pc3 new_pc4
          new_alu1 new_alu2 new_aluout new_aluout1 new_a new_b new_ir
          new_smdr new_pswo new_cond new_ra2 new_rb2 new_op2 new_op3
          new_op4 new_rd2 new_rd3 new_rd4 new_pa2 new_wipe new_clk)
```

Note that the register write-back which is the only action that executes in stage five of the pipeline is performed in phase one, as shown in section 4.2.4, and therefore does not appear in the `PHASE2_SPEC` definition.

Now that we have presented the two phase level instructions we can now define the `instructions` function for the phase level. For this level the `*key` is the clock variable `clk`. If the clock is high[2] then we execute `PHASE1_SPEC`, otherwise we execute `PHASE2_SPEC`.

```
⊢def ∀decrep orep mrep key.
    phase_inst decrep orep mrep key =
    (key → PHASE1_SPEC decrep orep mrep | PHASE2_SPEC decrep orep mrep)
```

4.2.7 The `select` Function

The `select` function takes as input a state and an environment and returns the key that the `instructions` function uses to select the next instruction. For the phase level the key is the value of the clock, so the function `Phase_select` returns the value of the clock from the current state.

```
⊢def ∀s e. Phase_Select s e = Clke s
```

4.2.8 The `output` Function

The output feature of the generic interpreter theory is not used in the specification of SAWTOOTH. Therefore, we define the `Phase_output` function to return the trivial type `one`.

```
⊢def ∀orep mrep key s e. Phase_Output orep mrep key s e = one
```

[2]In hardware terminology this is stated as being on the rising edge of the clock.

4.2.9 subout, substate, and subenv Functions

Since there is no data abstraction between the EBM and the phase level the **substate** function is defined to be the identity function I. The environment for the microprocessor is the same for all the levels in our interpreter hierarchy so the **subenv** function is defined to be the identity function also. Since we were not concerned with output there is no output abstraction. Thus, we defined the **subout** function to be I.

4.2.10 The Impl Function

The function **Impl** is the lower level interpreter which implements the current interpreter we are specifying. For the phase level, we used the EBM definition **sawtoothEBM**, defined in section 3.9, to implement the phase level.

4.2.11 The sync Function

The synchronization function is used by the generic interpreter theory to coordinate the temporal abstraction between the implementation level and the interpreter we are defining. Since each phase instruction executes in one phase of the clock, there is no temporal abstraction between the EBM and the phase level. Therefore the function **sync** was defined to be a function which returns **true**. This means that one time unit at the EBM level is the same as one time unit at the phase level.

```
⊢def ∀orep mrep s e. Phase_sync orep mrep s e = T
```

4.2.12 Instantiating the Interpreter

Now that we have defined concrete functions for all of the abstract functions of the generic interpreter theory we can now instantiate the theorem that describes the interpreter. The following specification of the phase level interpreter is created from the general interpreter specification in the GIT through instantiation [3].

```
⊢ ∀decrep orep mrep s e p.
    Phase_Interp decrep orep mrep s e p =
    (∀t.
      let k = Phase_Select(s t)(e t)
      in
       ((s(t + 1) = phase_inst decrep orep mrep k(s t)(e t)) ∧
        (p t = Phase_Output orep mrep k(s t)(e t))))
```

[3] The theorem **Phase_Interp** is not necessary to verify the phase level interpreter, however this theorem is necessary for verifying the higher level interpreters.

4.2.13 Summary

Table 4.2 shows a summary of the concrete definitions used to instantiate the generic interpreter theory for the phase level.

Abstract Function	*Concrete Definition*
`instructions`	`phase_inst`
`select`	`Phase_Select`
`output`	`Phase_output`
`subout`	`I`
`substate`	`I`
`subenv`	`I`
`Impl`	`sawtoothEBM`
`sync`	`Phase_sync`

Table 4.2: The abstract functions and their phase level definitions.

4.2.14 Theory Obligations

The verification of the phase level of SAWTOOTH is based on three assumptions:

1. During execution of the processor exceptions are not encountered.
2. The reset flag is never set.
3. Every instruction that is fetched is a valid instruction.

The **HOL** definition below shows how these assumptions were defined. These assumptions are made to simplify the top level verification. The reasoning behind these assumptions is that how could we verify an instruction if, for example, the processor recieves a reset signal during the five clock cycles that the instruction is being executed.

Each condition results in stalling the pipeline (inserting a NOOP), and stalling due to hazards is dealt with in great detail in the top level proof.

```
phase_assums =
⊢def ∀rep.
    phase_assums rep =
    (∀e1 t1. ¬Excp(e1 t1)) ∧
    (∀e1 t1. ¬Resetp(e1 t1)) ∧
    (∀op.
      op IN
      32,33,34,8,9,10,11,12,13,14,16,17,18,19,20,21,22,23,24,25,26,27,
       28,29,1,2,3,4,5,6,0,56,57,48,49,50,51,52,53,54,55)
```

4.3 Verifying the Phase Level Interpreter

To verify an interpreter we start with the goal shown below.

```
let GI_type = abs_type_info (theorem `GI_def` `GI`);;

let GI_REP =
"GI
 (phase_inst decrep orep mrep)
 Phase_Select
 (Phase_Output orep mrep)
 I
 I
 I
 (sawtoothEBM orep mrep decrep)
 (Phase_sync orep mrep)";;

g "IS_GI ^GI_REP";;
```

In the above goal `GI_type` is the abstract type of a generic interpreter. The term `GI_REP` is a term describing the interpreter representation we have just defined. The goal `IS_GI GI_REP` states that if this goal is proven then the interpreter we have defined satisfies the theory obligations for the generic interpreter theory: `INST_CORRECT` and `OUTPUT_CORRECT` [Win93].

Since we did not consider output in the specification of SAWTOOTH, the `OUTPUT_CORRECT` subgoal was trivial to prove. Proving the `INST_CORRECT` subgoal consisted of proving that the definition `sawtoothEBM`, from section 3.9, implied the two phase level instructions. For each phase level instruction, `PHASE1_SPEC` and `PHASE2_SPEC`, all of the EBM definitions inside of `sawtoothEBM` were expanded. Then the proof broke down to proving that each state element was updated in the same manner at both the EBM level and phase level. This broke the proof into 28 subgoals. Each of these subgoals is then proven with some rewriting. For the `PHASE1_SPEC` proof, this is quite simple since only the clock and register file are updated. However, the `PHASE2_SPEC` proof broke down to 26 subgoals. Although computationally intensive, the overall proof was mostly a matter of rewriting and not very difficult compared to the proofs of the architectural instructions.

After proving this goal we instantiate the correctness theorem.

```
GI_THOBS_THM=
..... ⊢ ∀decrep.
          IS_GI
          (GI
           (phase_inst decrep orep mrep)
           Phase_Select
           (Phase_Output orep mrep)
           I
           I
           I
           (sawtoothEBM orep mrep decrep)
           (Phase_sync orep mrep))



let PHASE_LEVEL_IS_CORRECT_aux =
  instantiate_abstract_theorem `GI_def` `IMPL_IS_CORRECT`
    [ "gi:^GI_type", GI_REP;
     ]
     [(SPEC_ALL GI_THOBS_THM)];;



PHASE_LEVEL_IS_CORRECT_aux =
..... ⊢ ∀s' e' p'.
          let s t = I(s' t)
          and e t = I(e' t)
          and p t = I(p' t)
          and f t = Phase_sync orep mrep(s' t)(e' t)
          in
           let abs = Temp_Abs f
           in
            (sawtoothEBM orep mrep decrep s' e' p' ∧ (∃t. f t) ⟹
             INTERP
             (GI
              (phase_inst decrep orep mrep)
              Phase_Select
              (Phase_Output orep mrep)
              I
              I
              I
              (sawtoothEBM orep mrep decrep)
              (Phase_sync orep mrep))
             (s o abs)
             (e o abs)
             (p o abs))
```

The above theorem can now be simplified with some rewriting.

```
PHASE_LEVEL_IS_CORRECT =
..... ⊢ ∀s' e' p'.
          sawtoothEBM orep mrep decrep s' e' p' ⟹
          Pipeline_Interp decrep orep mrep s' e' p'
```

We have presented the phase level verification of the SAWTOOTH microprocessor. At close inspection there is not much difference between the EBM definitions for SAWTOOTH and the phase level specifications. The reason for this is that the EBM definitions were written from a behavioral (rather than a structural) VHDL. Because of this the EBM specifications were written at a much higher level than other microprocessor specifications [Hun87, Win90b]. The actual verification was not difficult since it was mostly a matter or rewriting. However the importance of this verification is that gives us a verified phase level to use to create a pipeline level specification. The two phase instructions `PHASE1_SPEC` and `PHASE2_SPEC` will be combined to build the pipeline specification, and this verification assures us that our two building blocks are correct. After the pipeline level is specified we will verify that it correctly implements the top level instructions for the microprocessor.

4.4 The Pipeline Level Interpreter

In this section we present the specification for the pipeline level interpreter. The pipeline interpreter describes the actions that take place in the pipeline during one entire clock cycle. It is this specification that we will use to verify the macro level instructions.

4.4.1 The Pipeline State

The state of the microprocessor at the pipeline level is almost exactly the same as the state for the EBM level defined in section 3.3. The only difference is that the state variable clock is abstracted away.

4.4.2 The Pipeline Enviornment

The environment for the microprocessor at the pipeline level is the same as the environment at the EBM level as defined in section 3.3.1.

4.4.3 The `instructions` Function

There is only one pipeline instruction to execute at the pipeline level so our `instructions` functions is trivially simple. The pipeline instruction, `PIPELINE_SPEC`, was created by gluing together the two phase level instructions.

The important difference is the updating of the register file which takes place in the first phase of the clock cycle. In our pipeline instruction we take advantage of the semantics of the `let` statement in ML to model this behavior.

The first thing that is specified in the definition for `PIPELINE_SPEC` is that the register file is updated with the write-back. From this point, the rest of `PHASE2_SPEC` is simply tacked onto the end. However, where ever the state element `reg` was referenced in `PHASE2_SPEC`, it is replaced with `new_reg`. This models the actions taking place in phase two are reading from a register file that was updated correctly in phase one.

```
PIPELINE_SPEC =
⊢def ∀decrep orep mrep s e.
    Pipeline_SPEC decrep orep mrep s e =
    (let reg = RegP s
     and psw = PswP s
     and pc1 = Pc1P s
     and mem = MemP s
     and pc2 = Pc2P s
     and pc3 = Pc3P s
     and pc4 = Pc4P s
     and alu1 = Alu1P s
     and alu2 = Alu2P s
     and aluout = AluoutP s
     and aluout1 = Aluout1P s
     and a = APi s
     and b = BPi s
     and ir = IrP s
     and smdr = SmdrP s
     and pswo = PswoP s
     and cond = CondP s
     and ra2 = Ra2P s
     and rb2 = Rb2P s
     and op2 = Op2P s
     and op3 = Op3P s
     and op4 = Op4P s
     and rd2 = Rd2P s
     and rd3 = Rd3P s
     and rd4 = Rd4P s
     and wipe = WipeP s
     and pa2 = Pa2P s
     and exc = Excp e
     and ivec = Ivecp e
     and reset = Resetp e
     in
      let new_reg = UPDATE_REG orep rd4 reg aluout1 in ...
```

The `instructions` function itself is simply defined to return the instruction `Pipeline_SPEC` regardless of the value of the `key`.

```
⊢def ∀decrep mrep orep key.
    pipeline_inst decrep orep mrep key = Pipeline_SPEC decrep orep mrep
```

4.4.4 The `select` Function

Since there is only one pipeline instruction, the `select` instruction is trivially defined to return `one`.

```
⊢def ∀s e. pipeline_select s e = one
```

4.4.5 The Output Function

The output function is once again trivially defined.

```
⊢def ∀orep mrep key s e. pipeline_output orep mrep key s e = one
```

4.4.6 subout,substate, and subenv Functions

Both **subenv** and **subout** are once again defined to be I. However we do have some data abstraction. The state variable **clk** is abstracted away at the pipeline level. The definition **Pipeline_Substate** shown below relates the state at the phase level to the state at the pipeline level.

```
Pipeline_Substate =
⊢def ∀s.
    Pipeline_Substate s =
    (let reg = Rege s
     and psw = Pswe s
     and pc1 = Pc1e s
     and mem = Meme s
     and pc2 = Pc2e s
     and pc3 = Pc3e s
     and pc4 = Pc4e s
     and alu1 = Alu1e s
     and alu2 = Alu2e s
     and aluout = Aluoute s
     and aluout1 = Aluout1e s
     and a = Ae s
     and b = Be s
     and ir = Ire s
     and smdr = Smdre s
     and pswo = Pswoe s
     and cond = Conde s
     and ra2 = Ra2e s
     and rb2 = Rb2e s
     and op2 = Op2e s
     and op3 = Op3e s
     and op4 = Op4e s
     and rd2 = Rd2e s
     and rd3 = Rd3e s
     and rd4 = Rd4e s
     and pa2 = Pa2e s
     and wipe = Wipee s
     in
      Pipeline_State reg psw pc1 mem pc2 pc3 pc4 alu1 alu2 aluout
      aluout1 a b ir smdr pswo cond ra2 rb2 op2 op3 op4 rd2 rd3
      rd4  pa2  wipe)
```

4.4.7 The Impl Function

The function that implements the pipeline level interpreter is the phase level interpreter, **Phase_Interp**.

```
⊢ ∀decrep orep mrep s e p.
    Phase_Interp decrep orep mrep s e p =
    (∀t.
      let k = Phase_Select(s t)(e t)
      in
       ((s(t + 1) = phase_inst decrep orep mrep k(s t)(e t)) ∧
        (p t = Phase_Output orep mrep k(s t)(e t))))
```

4.4.8 The sync Function

Between the phase level and the pipeline level there is some temporal abstraction. The unit of time at the phase level is one phase of the clock, and the unit of time at the pipeline level is one clock cycle. In other words, it takes two phase level instructions to execute one pipeline instruction. The two levels are synchronized when the clock is high and the phase level is about to execute the phase one instruction. From this, we define our `sync` function as simply the value of the clock.

```
⊢def ∀s e. pipeline_sync s e = Clke s
```

4.4.9 Summary

Table 4.3 shows the concrete definitions that we use to instantiate the generic interpreter theory. The instantiated interpreter theorem `Pipeline_Interp` is also shown below.

```
⊢ ∀decrep orep mrep s e p.
    Pipeline_Interp decrep orep mrep s e p =
    (∀t.
      let k = pipeline_select(s t)(e t)
      in
       ((s(t + 1) = pipeline_inst decrep orep mrep k(s t)(e t)) ∧
        (p t = pipeline_output orep mrep k(s t)(e t))))
```

Abstract Function	*Concrete Definition*
instructions	pipeline_inst
select	pipeline_select
output	pipeline_output
subout	I
substate	Pipeline_Substate
subenv	I
Impl	Phase_Interp
sync	pipeline_sync

Table 4.3: The abstract functions and their pipeline level definitions.

The verificaiton of the pipeline level was carried in the same manner as the phase level using the generic interpreter theory. The correctness lemma for pipeline level is shown below. The importance of the pipeline verification is that the pipeline interpreter specification will be used verify the top level instructions, and this verification gives us some confidence that we are starting with a correct representation of the what the hardware is doing.

```
PIPELINE_LEVEL_IS_CORRECT =
... ⊢ ∀s′ e′ p′.
        Phase_Interp decrep orep mrep s′ e′ p′ ∧
        (∃t. pipeline_sync(s′ t)(e′ t)) ⟹
        Pipeline_Interp
        decrep
        orep
        mrep
        (λ x.
          Pipeline_Substate
          (s′(Temp_Abs(λ t. pipeline_sync(s′ t)(e′ t))x)))
        (λ x. e′(Temp_Abs(λ t. pipeline_sync(s′ t)(e′ t))x))
        (λ x. p′(Temp_Abs(λ t. pipeline_sync(s′ t)(e′ t))x))
```

Chapter 5

The Architectural Level

The architectural level specifies what the assembly language programmer sees. For the specification of the top level we once again use the generic interpreter theory. However, the verification of this level will not be done in the manner used to verify the phase and pipeline levels. We start once again by defining the state of the processor.

5.1 The State

There is much data abstraction between the pipeline specification and the architectural specification. The state of the processor at this level only contains five elements:

- `reg`: The register file.
- `mem`: Memory.
- `pc`: The program counter.
- `psw`: The program status word.
- `pa`: The page address flag.

5.2 The Environment

The environment at the architectural level is the same as the environment at all the previous levels specified.

5.3 The instructions Function

5.3.1 The Instructions

At the top level there are currently 41 instructions implemented. The syntax for the assembly language instructions is defined by creating an abstract instruction syntax as described by Windley[Win94, page 4]. The abstract syntax for SAWTOOTH is shown in Figure 5.1.

ALU Instructions The next step in specifying the assembly instructions is to write a concrete function for each instruction which specifies their state transition behavior. We will not present the specification for every instruction here, but we present one instruction from each of the major categories. We start with an ALU instruction–the ADD instruction.

```
M_ADD =
⊢def ∀orep mrep Rd Ra Rb s e.
    M_ADD orep mrep Rd Ra Rb s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s in
      let a = FETCH_REG orep reg Ra pa2   in
       let b = FETCH_REG orep reg Rb pa2  in
        let result = add orep(a,b)  in
         let cflag = addp orep(a,b)
         and vflag = aovfl orep(a,b)  in
          let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
          and new_psw = (psw except [vf,vflag;cf,cflag])orep
          and new_pc = inc orep pc in
            Macro_State new_reg new_psw new_pc mem pa2)
```

The specification for each instruction is written in a denotational style. The function `M_ADD` defines the behavior of the ADD instruction. `M_ADD` takes five arguments: the destination register, the first source register, the second source register, a state, and an environment. All three of the registers are of type `*word5` as per the opcode format for the ADD instruction. However the page address bit is added to the register indexes to create the six bit register addressed needed by the register file. The appending of the page address bit to the two source registers and the destination register is done by the two functions `FETCH_REG` and `MACRO_UPDATE_REG` [1] shown in Appendix C.

The specification of the ADD instruction is fairly straightforward. The two source registers are fetched from the register file, the two operands are added together, the overflow and carry flags are calculated, the destination register is written to with the resulting value from the ADD, the program status word is updated by setting the new carry and overflow flags, and the program counter is is incremented.

[1] The term `macro` is a holdover from previous verification efforts [Win90b].

```
let Instruction_axiom = define_type `Instruction`
   `Instruction =
    LD    *ri *ri *imm                                          |
    LHI   *ri *ri *imm                                          |
    ST    *ri *ri *imm                                          |
    ADDI  *ri *ri *imm                                          |
    ADDCI *ri *ri *imm                                          |
    SUBI  *ri *ri *imm                                          |
    SUBCI *ri *ri *imm                                          |
    BANDI *ri *ri *imm                                          |
    BORI  *ri *ri *imm                                          |
    BXORI *ri *ri *imm                                          |
    ADD   *ri *ri *ri                                           |
    ADDC  *ri *ri *ri                                           |
    SUB   *ri *ri *ri                                           |
    SUBC  *ri *ri *ri                                           |
    BAND  *ri *ri *ri                                           |
    BOR   *ri *ri *ri                                           |
    BXOR  *ri *ri *ri                                           |
    BNOT  *ri *ri                                               |
    LSL   *ri *ri                                               |
    LSR   *ri *ri                                               |
    ASR   *ri *ri                                               |
    BSL   *ri *ri                                               |
    BSR   *ri *ri                                               |
    ABSR  *ri *ri                                               |
    SPA   *imm                                                  |
    RPA   *ri                                                   |
    GPSW  *ri                                                   |
    PPSW  *ri                                                   |
    INT   *ri *ri *imm                                          |
    RTI   *ri *imm                                              |
    NOOP                                                        |
    JMP   *word26                                               |
    JALI  *ri *ri *imm                                          |
    BEQ   *ri *ri *imm                                          |
    BNEQ  *ri *ri *imm                                          |
    BGTZ  *ri *imm                                              |
    BGTEZ *ri *imm                                              |
    BLTZ  *ri *imm                                              |
    BLTEZ *ri *imm                                              |
    BC    *imm                                                  |
    BOV   *imm
                                             `;;
```

Figure 5.1: The Abstract Syntax for the Instructions

ALU Immediate Instructions The next class of instructions we consider is the ALU Immediate Instructions. The specification for the ADDI instruction is shown below.

```
M_ADDI =
⊢def ∀orep mrep Rd Ra imm s e.
    M_ADDI orep mrep Rd Ra imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s
     in
     let a = FETCH_REG orep reg Ra pa2
         and i = extend orep imm  in
     let result = add orep(a,i)   in
     let cflag = addp orep(a,i)
        and vflag = aovfl orep(a,i)   in
     let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
         and new_psw = (psw except [vf,vflag;cf,cflag])orep
         and new_pc = inc orep pc  in
          Macro_State new_reg new_psw new_pc mem pa2)
```

The only difference between ADDI and the normal ADD is that the two operands for the ADDI instruction are the contents of the `Ra` register and the 16-bit immediate value as opposed to adding the contents of two registers.

Shift Instructions Besides ALU and ALUI instructions SAWTOOTH also has six shift instructions. The specification for the Logical Shift Left (LSL) instruction is shown below.

```
M_LSL =
⊢def ∀orep mrep Rd Ra s e.
    M_LSL orep mrep Rd Ra s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let a = FETCH_REG orep reg Ra pa2 in
       let result = shl orep(a,1)  in
        let cflag = msb orep a in
         let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
         and new_psw = (psw except [cf,cflag])orep
         and new_pc = inc orep pc in
          Macro_State new_reg new_psw new_pc mem pa2)
```

The shift instructions execute like most ALU instructions except they take only one operand–the register `Ra`. The result of the LSL instruction is that the destination register is updated with the value of the `Ra` register shifted left one bit. The carry flag in the program status word is also updated with the value of the most significant bit that was shifted out.

Branch Instructions SAWTOOTH has ten branch instructions. As an example we present the specification of the Branch when Equal instruction, BEQ.

```
M_BEQ =
⊢def ∀orep mrep Rd Ra imm s e.
    M_BEQ orep mrep Rd Ra imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let rs1 = FETCH_REG orep reg Ra pa2 in
       let rs2 = FETCH_REG orep reg Rd pa2  in
        let a = inc orep pc  in
         let i = signextend orep imm  in
          let result = add orep(a,i) in
           let new_pc = (eqp orep(rs1,rs2) →  result | inc orep pc)  in
            Macro_State reg psw new_pc mem pa2)
```

The BEQ instruction branches execution to the branch target if the contents of register `Rd` equal the contents of register `Ra`. The branch target is calculated by adding the contents of `Ra` to the immediate.

Load and Store Instructions Like most RISC machines, SAWTOOTH uses a standard load-store architecture. To move data to and from memory Sawtooth has three load-store instructions. The specification for the LOAD instruction is shown below.

```
M_LD =
⊢def ∀orep mrep Rd Ra imm s e.
    M_LD orep mrep Rd Ra imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
    let a = FETCH_REG orep reg Ra pa2
      and i = signextend orep imm  in
    let result = fetch mrep(mem,address orep(add orep(a,i)))  in
    let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
        and new_pc = inc orep pc in
    let new_pa2 = pa2 in Macro_State new_reg psw new_pc mem new_pa2)
```

The LOAD instruction loads a word from memory. The address to fetch from memory is calculated by adding the contents of register `Ra` to the immediate value. The word fetched from memory is stored into the destination register `Rd`.

Special Instructions Finally, SAWTOOTH has seven special instructions which perform various functions. Below, we show the specification for the Set Page Address instruction.

```
M_SPA =
⊢def ∀orep mrep imm s e.
    M_SPA orep mrep imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let new_pa = bit orep lsbv(extend orep imm)  in
       let new_psw =
             (psw except [pa,bit orep lsbv(extend orep imm)])orep  in
        let new_pc = inc orep pc in
         Macro_State reg new_psw new_pc mem new_pa)
```

The SPA instruction sets the page address flip-flop to the value of the lowest bit (`lsbv`) of the immediate. The program status word is also updated to show the change in the page address bit.

A complete listing of the top level instruction definitions are in appendix D.

5.3.2 Putting Them Altogether

After specifying every instruction, we combine these definitions into one which defines the semantics for any instruction. This function, `M_INST` is shown in Figure 5.2.

Now it looks as though this could be our `instructions` function for the generic interpreter theory. However, SAWTOOTH also handles external interrupts. The specification of an external interrupt is also done in the denotational style as above [2].

```
⊢ ∀orep mrep s e.
    M_EINT orep mrep s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s
     in
      let a = FETCH_REG orep reg R0 pa2 in
      let i = extend orep(low16w orep(Ivecp e)) in
      let result = band orep(add orep(a,i),wordn orep 255) in
      let new_psw =
              ((shl orep(psw,8)) except
               [pa,get_pa orep psw;sm,T;ie,F;cf,get_cf orep psw;
                vf,get_vf orep psw])
              orep  in
      let new_reg =
               MACRO_UPDATE_REG orep reg R1 pa2 psw(inc orep pc) in
      let new_pc = result in
      let new_pa = get_pa orep psw in
            Macro_State new_reg new_psw new_pc mem new_pa)
```

[2] This definition will prove to be inadequate, when we try to prove a correctness theorem about interrupts. It is left here for two reasons: to maintain completeness of the chapter and to highlight what changes had to be made the specification due to the architecture used to implement interrupts. For a more thorough discussion of this subject, please see section 6.5

M_INST =
$\vdash_{def}$ (∀op m Rd Ra imm. M_INST op m(LD Rd Ra imm) = M_LD op m Rd Ra imm) ∧
(∀op m Rd Ra imm. M_INST op m(LHI Rd Ra imm) = M_LHI op m Rd Ra imm) ∧
(∀op m Rd Ra imm. M_INST op m(ST Rd Ra imm) = M_ST op m Rd Ra imm) ∧
(∀op m Rd Ra imm. M_INST op m(ADDI Rd Ra imm) = M_ADDI op m Rd Ra imm) ∧
(∀op m Rd Ra imm. M_INST op m(ADDCI Rd Ra imm) = M_ADDCI op m Rd Ra imm) ∧
(∀op m Rd Ra imm. M_INST op m(SUBI Rd Ra imm) = M_SUBI op m Rd Ra imm) ∧
(∀op m Rd Ra imm. M_INST op m(SUBCI Rd Ra imm) = M_SUBCI op m Rd Ra imm) ∧
(∀op m Rd Ra imm. M_INST op m(BANDI Rd Ra imm) = M_BANDI op m Rd Ra imm) ∧
(∀op m Rd Ra imm. M_INST op m(BORI Rd Ra imm) = M_BORI op m Rd Ra imm) ∧
(∀op m Rd Ra imm. M_INST op m(BXORI Rd Ra imm) = M_BXORI op m Rd Ra imm) ∧
(∀op m Rd Ra Rb. M_INST op m(ADD Rd Ra Rb) = M_ADD op m Rd Ra Rb) ∧
(∀op m Rd Ra Rb. M_INST op m(ADDC Rd Ra Rb) = M_ADDC op m Rd Ra Rb) ∧
(∀op m Rd Ra Rb. M_INST op m(SUB Rd Ra Rb) = M_SUB op m Rd Ra Rb) ∧
(∀op m Rd Ra Rb. M_INST op m(SUBC Rd Ra Rb) = M_SUBC op m Rd Ra Rb) ∧
(∀op m Rd Ra Rb. M_INST op m(BAND Rd Ra Rb) = M_BAND op m Rd Ra Rb) ∧
(∀op m Rd Ra Rb. M_INST op m(BOR Rd Ra Rb) = M_BOR op m Rd Ra Rb) ∧
(∀op m Rd Ra Rb. M_INST op m(BXOR Rd Ra Rb) = M_BXOR op m Rd Ra Rb) ∧
(∀op m Rd Ra. M_INST op m(BNOT Rd Ra) = M_BNOT op m Rd Ra) ∧
(∀op m Rd Ra. M_INST op m(LSL Rd Ra) = M_LSL op m Rd Ra) ∧
(∀op m Rd Ra. M_INST op m(LSR Rd Ra) = M_LSR op m Rd Ra) ∧
(∀op m Rd Ra. M_INST op m(ASR Rd Ra) = M_ASR op m Rd Ra) ∧
(∀op m Rd Ra. M_INST op m(BSL Rd Ra) = M_BSL op m Rd Ra) ∧
(∀op m Rd Ra. M_INST op m(BSR Rd Ra) = M_BSR op m Rd Ra) ∧
(∀op m Rd Ra. M_INST op m(ABSR Rd Ra) = M_ABSR op m Rd Ra) ∧
(∀op m imm. M_INST op m(SPA imm) = M_SPA op m imm) ∧
(∀op m Rd. M_INST op m(RPA Rd) = M_RPA op m Rd) ∧
(∀op m Rd. M_INST op m(GPSW Rd) = M_GPSW op m Rd) ∧
(∀op m Rd. M_INST op m(PPSW Rd) = M_PPSW op m Rd) ∧
(∀op m Rd Ra imm. M_INST op m(INT Rd Ra imm) = M_INT op m Rd Ra imm) ∧
(∀op m Rd imm. M_INST op m(RTI Rd imm) = M_RTI op m Rd imm) ∧
(∀op m. M_INST op m NOOP = M_NOOP op m) ∧
(∀op m imm26. M_INST op m(JMP imm26) = M_JMP op m imm26) ∧
(∀op m Rd Ra imm. M_INST op m(JALI Rd Ra imm) = M_JALI op m Rd Ra imm) ∧
(∀op m Rd Ra imm. M_INST op m(BEQ Rd Ra imm) = M_BEQ op m Rd Ra imm) ∧
(∀op m Rd Ra imm. M_INST op m(BNEQ Rd Ra imm) = M_BNEQ op m Rd Ra imm) ∧
(∀op m Rd imm. M_INST op m(BGTZ Rd imm) = M_BGTZ op m Rd imm) ∧
(∀op m Rd imm. M_INST op m(BGTEZ Rd imm) = M_BGTEZ op m Rd imm) ∧
(∀op m Rd imm. M_INST op m(BLTZ Rd imm) = M_BLTZ op m Rd imm) ∧
(∀op m Rd imm. M_INST op m(BLTEZ Rd imm) = M_BLTEZ op m Rd imm) ∧
(∀op m imm. M_INST op m(BC imm) = M_BC op m imm) ∧
(∀op m imm. M_INST op m(BOV imm) = M_BOV op m imm)

Figure 5.2: The Meaning of any one instruction

When an external interrupt occurs the program status word is shifted left 8 bits, the value of the program counter incremented is stored in the R1 register, and the program counter receives the value of the binary and of the value 255 with the interrupt vector. In effect, this sets the lower eight bits of the program counter.

We can now combine the definition of M_INST and M_EINT to create the instructions function.

```
⊢def ∀orep mrep key.
    arch_inst orep mrep key =
    (ISL key →  M_INST orep mrep(OUTL key) | M_EINT orep mrep)
```

Now this definition looks bizarre. The type of the *key for the GIT is Inst_type + one. This is a coproduct type where the type one specifies that an external interrupt has occurred. This key is passed to the instructions function by the function Opcode shown in the next section.

5.4 The select Function

At the top level, the select function either returns a decoded instruction or a key of type one which signifies an interrupt. An interrupt can only occur if the interrupt flag from the environment is high and interrupts are currently enabled. The function get_ie returns the value of the interrupt enable bit of the program status word.

```
Opcode =
⊢def ∀decrep orep mrep s e.
    Opcode orep mrep decrep s e =
    (let psw = Psw s
     and pc = Pc s
     and mem = Mem s
     in
      ((Excp e ∧ get_ie orep psw) →
       INR one |
       INL(decode_word decrep(fetch mrep(mem,address orep pc)))))
```

5.5 The output Function

As for all of the specified levels in SAWTOOTH, the output function is trivially defined.

```
Arch_Output =
⊢def ∀orep mrep key s e. Arch_Output orep mrep key s e = one
```

5.6 The `subout`, `substate`, and `subenv` Functions

There is no data abstraction between the pipeline and top level environments, so the **`subenv`** function is trivially defined to be `I`. There is also no abstraction between output at the various levels so the **`subout`** function is also trivially defined.

However, between the pipeline level state and the architectural level state there is a a great deal of data abstraction. The function `Macro_Substate` defined below performs the data abstraction between the pipeline and phase level interpreters.

```
Arch_Substate =
⊢def ∀s.
    Arch_Substate s =
    (let reg = RegP s
     and psw = PswP s
     and pc = Pc1P s
     and mem = MemP s
     and pa2 = Pa2P s
     in
      Macro_State reg psw pc mem pc pa2)
```

5.7 The `Impl` Function

The pipeline level interpreter is the implementation for the macro level, therefore the instantiated theorem `Pipeline_Interp` (see section 4.4.9) will be the implementation for the top level interpreter.

5.8 The `sync` Function

The synchronization function is used to define when two interpreters are synchronized in time. However, in a pipelined microprocessor several instructions are executing concurrently. Because of this the data and temporal abstraction occur at the same time and a simple synchronization function can not be used to model the abstraction that is actually occurring. Thus the generic interpreter theory is not used in the verification of the architectural level. However to complete the specification we trivially define a synchronization function.

```
⊢def ∀s e. arch_sync s e = T
```

5.9 Summary

Table 5.1 shows a summary of the concrete definitions used to instantiate the architectural level interpreter. The instantiated theorem describing the structure of the interpreter is also shown below. The verification of the top level will be discussed in the next chapter.

```
⊢ ∀decrep orep mrep s e p.
    Arch_Interp decrep orep mrep s e p =
    (∀t.
      let k = Opcode orep mrep decrep(s t)(e t)
      in
       ((s(t + 1) = arch_inst orep mrep k(s t)(e t)) ∧
        (p t = Arch_Output orep mrep k(s t)(e t))))
```

Abstract Function	*Concrete Definition*
`instructions`	`arch_inst`
`select`	`Opcode`
`output`	`Arch_Output`
`subout`	`I`
`substate`	`Arch_Substate`
`subenv`	`I`
`Impl`	`Pipeline_Interp`
`sync`	`arch_sync`

Table 5.1: The abstract functions and their top level definitions.

Chapter 6

The Top Level Verification

In this chapter we present the results for our top level verification efforts for the SAWTOOTH microprocessor. As stated earlier the verification was not done using the generic interpreter theory–even though the macro level was specified with the generic interpreter theory. A full description of why we do this is presented in section 6.3.

The verification was done by showing that the pipeline interpreter correctly implements the architectural level instructions. We stated earlier that this verification does not make any large assumptions about correct instruction pipelining with respect to hazards. However, there are some assumptions that had to be made in order for the proof to be valid. We start this chapter be describing these assumptions.

6.1 Necessary Assumptions

6.1.1 No Self-Modifying Code

For SAWTOOTH, the processor is not assumed to be attached to a cache system that caches instructions separately from data. Therefore, user memory is one piece of state as opposed to having a state element for instruction memory and data memory as was done in the Uinta proof[WC94]. This leads to a problem that instructions in the pipeline could write to a portion of memory where instructions are stored and affect the execution of future instructions. This makes it difficult to verify instructions because we don't know if a previous instruction has overwritten or changed an instruction that is already in the pipeline. In order to avoid this problem we assume that store instructions do not modify the instruction stream:

```
memory_assump =
⊢ ∀orep mrep.
    memory_assump orep mrep =
    (∀t mem s w v.
      fetch
      mrep
      (store mrep(mem,address orep w,v),address orep(Pc1P(s t))) =
      fetch mrep(mem,address orep(Pc1P(s t))))
```

This assumptions says that if we are going to fetch an instruction from a memory that has been updated from a store, the results of the instruction fetch will be the same as if we fetched from the memory prior to the store. This condition arises when we stall because a load or store instruction is writing (or reading) from memory, and at the next time unit we want to know that the instruction we are fetching from the updated memory is the same as the one we should have fetched prior to the stall.

Note however, that the assumption is valid only for memory locations that are covered by the program counter, thus memory will change but the instructions themselves will not.

6.1.2 The PPSW Assumption

Our verification turned up some errors in the hardware design of the microprocessor. Unfortunately two of these errors were found very late in the design process and could not be changed. One of the errors we discovered occurs because of the PPSW instruction, and the other deals with the LOAD instruction.

In the specification (and VHDL) for the mux that loads the destination register `rd2`, the logic checks to see if the processor is in supervisory mode. It does that by checking the supervisory mode bit in the program status word. Now this check is made in stage 2 of the pipeline for any given instruction. However most instructions change the program status word in stage three after the ALU processes the instruction. So, for a two instruction sequence of assembly code, the second instruction is not checking the correct program status word to see if the processor is in supervisory mode since the previous instruction has yet to update the program status word.

Fortunately for the designer, only three instructions change the value of the supervisory mode bit: Interrupt (INT), Return from Interrupt (RTI), and Put Program Status Word. After a INT or RTI instruction is fetched the pipeline is stalled, so our problem is solved there. However the processor does not stall after a PPSW instruction. A code segment like the one in Figure 6.1 leads to an error if the processor is in supervisory mode. The PPSW instruction updates the program status word in its stage three. The new program status word is available to read at the end of the third stage which is marked with an asterisk. However the ADD instruction checks the supervisory mode bit of the program status word in its stage two, and therefore in this example it is not reading from the current program status word.

PPSW	Ra	IF	ID	EX *	MEM	WB	
ADD	Rd Rs1 Rs2		IF	* ID	EX	MEM	WB

Figure 6.1: PPSW Error

Since this error was discovered late, we overcome this error by creating a software requirement that a NOOP instruction must follow a PPSW. We define this assumption below.

```
ppsw_assump =
⊢def ∀orep decrep mrep s.
    ppsw_assump orep decrep mrep s =
    (∀t.
      (FST(Wordn_decode orep decrep(IrP(s t))) = 4) ⟹
      (decode_word decrep(fetch mrep(MemP(s t),address orep(Pc1P(s t)))) =
       NOOP))
```

The assumption states that if the opcode of the instruction in the instruction register equals that of a PPSW instruction, then the next instruction to be fetched from memory is a NOOP instruction.

6.1.3 The Load Assumption

Another error that was uncovered during the verification concerns an instruction immediately following a load referencing the destination register for the load instruction. In this case the resulting reference would result in the wrong value being forwarded to the next instruction. In many pipeline designs, a stall is inserted after load instructions to prevent this error, but for SAWTOOTH the designer erroneously chose not to stall.

LOAD	R3 R4 Imm	IF	ID	EX *	MEM	WB	
ADD	Rd R3 Rs2		IF	* ID	EX	MEM	WB

Figure 6.2: LOAD Error

Figure 6.2 shows a sequence of code which causes the load error. The value of register `R3` will not be available until the end of stage four of the load instruction where it is loaded into the `aluout1` register for immediate forwarding. At first glance, it would appear that data forwarding would solve this problem. Since the register `R3` is used as both a source register for the ADD and a destination register for the LOAD the the forwarding mux `mx4f` will forward the value of the `aluout` register to the ALU. However, the value of the `aluout` register is the output of the ALU in the previous stage which is the effective address in memory from where to fetch the contents of `R3`. In other words, the wrong value is being forwarded to the ALU.

Once again, the problem can be solved by inserting a stall after the LOAD instruction. So we make the same assumption about the LOAD that we did for the PPSW instruction.

```
load_assump =
⊢def ∀orep decrep mrep s.
    load_assump orep decrep mrep s =
    (∀t.
      (FST(Wordn_decode orep decrep(IrP(s t))) = 32) ⟹
      (decode_word decrep(fetch mrep(MemP(s t),address orep(Pc1P(s t)))) =
       NOOP))
```

This assumption states that if the instruction being decoded in stage two is a load then the instruction that is to be fetched is a NOOP instruction.

6.2 The Data Forwarding Lemma

As stated earlier, our verification is done by proving that the pipeline interpreter correctly implements the top level instruction specifications. Part of this goal is to prove that the register file is correctly updated by the pipeline. In order to prove this goal we needed the theorem shown below.

```
forwarding_lemma1 =
⊢ ∀decrep s e p t x1.
1    phase_assums orep ⟹
2    Pipeline_Interp decrep orep mrep s e p ⟹
3    ¬(Op2P(s(t + 1))) IN {5,6,56,57,54,48,51,50,53,52,49,55} ⟹
4    ¬(Op2P(s(t + 1))) IN {32} ⟹
5    ¬((Op3P(s t)) IN {54,48,51,50,53,52,49,55} ∧ CondP(s t)) ⟹
6    ¬(Op3P(s t)) IN {57,5,6} ⟹
7    ¬(Op3P(s t)) IN {32,34} ⟹
8    ((Data_hazard orep x1(Rd3P(s((t + 1) + 1))) →
9      AluoutP(s((t + 1) + 1)) |
10      (Data_hazard orep x1(Rd4P(s((t + 1) + 1))) →
11       Aluout1P(s((t + 1) + 1)) |
12       RegP(s((t + 1) + 1))x1)) =
13     RegP(s((((t + 1) + 1) + 1) + 1))x1)
```

This theorem states that data forwarding works correctly. The first few assumptions, lines 3-7, state that no stalls occurred during the time frame we are considering. The theorem states that for any register index x1:

- If there are no data hazards between x1 and the two previous instructions (lines 8 and 10 above), then the value in the register indexed by x1 at time $t + 2$ (line 12) will be the same as the value in the same register at time $t + 4$.

- However if there is a data hazard between x1 and one of the two previous instructions then the data forwarding hardware provides the correct value for the register file (lines 9 and 11).

The small theorem is vital to verifying the top level instructions. It is used to show that the register file is updated correctly by the value that is calculated by the ALU in stage three (if the instruction is an ALU instruction of course).

The proof of this theorem also lead directly to the discovery of the design error described in section 6.1.3. The error was discovered by trying to weaken the above theorem removing the assumption on line four of the theorem. The data forwarding lemma could not be proven if the instruction prior to the one we and considering is a LOAD.

6.3 Verifying an Instruction

The top level verification of SAWTOOTH begins by showing that the pipeline interpreter implements the assembly language instructions. Since most instructions execute in a very similar manner in the pipeline, we will concentrate here on the proof that one instruction is implemented correctly. The instruction we will focus on is the ADD instruction.

As stated in previous sections we do not use the generic interpreter theory in the verification of the top level specifications. To understand fully the top level verification of instructions, we must first look at why the generic interpreter theory fails in the verification of pipelined microprocessors. We start by examining how microprocessor verifications were completed for non-pipelined machines.

6.3.1 Why the Generic Interpreter Theory Fails

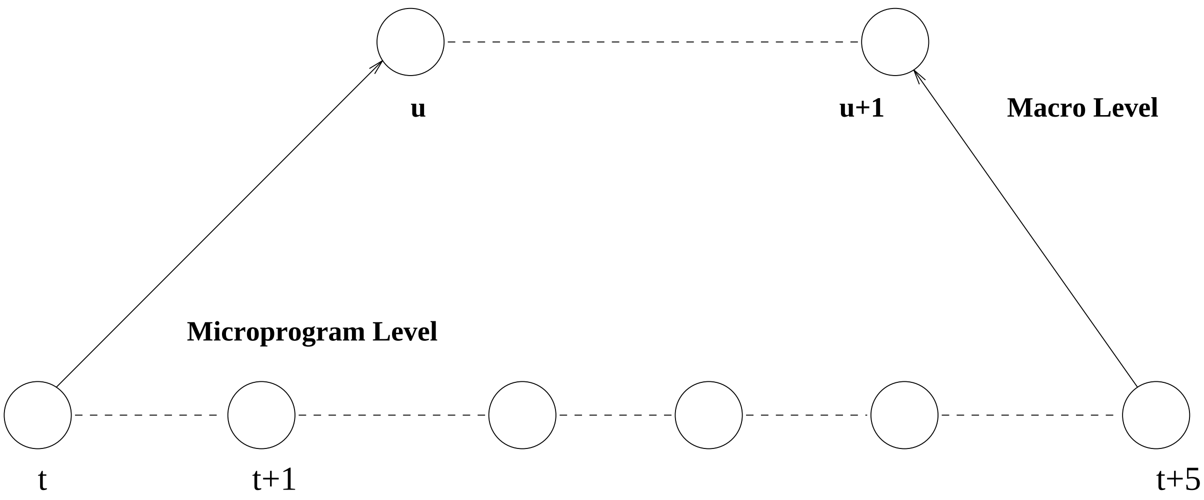

Figure 6.3: Data and Temporal Abstraction in Earlier Verification Efforts

Many of the earlier microprocessor verification efforts were microprogrammed processors [Joy89, Win90b]. Figure 6.3 shows the relationship between the microprogram level interpreter and the top level specification for a typical microprogrammed processor. At the top level one unit of time, u, denotes the length of time it takes for one assembly language instruction to execute. However, at the microprogram level a unit of time, t, is the amount of time it takes to execute one microinstruction. The difference between u and t is temporal abstraction. To verify an instruction the states of the two specifications are compared at some synchronization points. For example, a synchronization point would be when the microprogram counter equals zero representing the start of a new instruction. At these synchronization points, a data abstraction function is used to compare the states of the microprogram level and the macro level. The important thing to notice in the picture is that the the data abstraction and temporal abstraction are orthogonal to each other. We can start both levels in a common state, apply the temporal abstraction function, and then apply the data abstraction function to compare the next states of the two specifications. However, in a pipelined microprocessor the data and temporal abstraction occur at the same time.

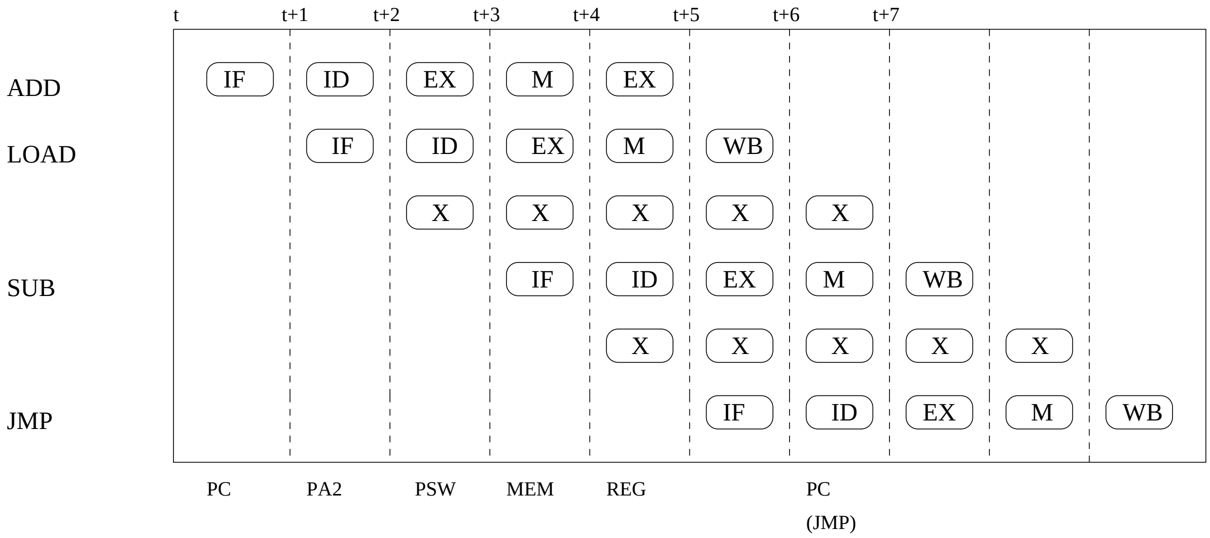

Figure 6.4: Data and Temporal Abstraction in the Pipeline

To discuss the correctness of a pipelined instruction in SAWTOOTH, we consider the following fragment of assembly code:

```
ADD   R4 R8 R7
LOAD  R8 R3  Imm
SUB   R5 R9  R4
JMP   Imm2
```

Figure 6.4 shows the pipelined instruction stream for this code. The top of the diagram labels the time unit in which each stage is executed. The left-hand side of Figure labels the instructions. The bottom of the diagram labels what state element is updated in the specified time unit for the ADD instruction (except for the far right PC label which signifies when the program counter is updated for the JMP instruction). This diagram highlights some interesting points about verifying pipelined instructions.

- The first thing to notice is that the five macro state elements are updated at different time intervals. This characteristic of pipelined architectures is called *skew*[WC94]. We can no longer just compare the macro state and the pipeline state and times t and $t + 5$ because during that interval, for example, the program counter can be updated up to 5 times as opposed to the one time in our top level specifications.

- Not only are the macro level state transitions spread out over time, the time intervals where these transitions occur is not always consistent among all instructions. For example, the ADD instruction updates the program counter, PC, between times t and $t + 1$. However, the JMP instruction updates the program counter between times $t + 1$ and $t + 2$. This phenomena is called *shifting*[WC94].

- Instructions may be delayed from executing due to hazards in the pipeline. In Figure 6.4, the JMP instruction is preceded by a NOOP. This NOOP is inserted because there is a *structural hazard*[HP90, page 258] between the LOAD instruction and the JMP instruction. Since there is only one bus to memory, the instruction fetch for the JMP must be *stalled* since the LOAD instruction must use the bus to load from memory.

Because of skewing, stalling and shifting, data and temporal abstraction in the pipeline model can not be separated, and therefore the generic interpreter theory can not be used to verify pipelined instructions.

As with the case of the Uinta verification [WC94], we must define a temporal abstraction function that maps pipeline state elements from different times into one macro level state record. However, before we do this we must examine more closely the effects branches have on the pipeline.

6.3.2 Branches

Conditional and unconditional branches causing *control hazards*[HP90, page 270] cause tremendous performance loss in the pipeline. They also cause us to re-think how instructions should be verified in our top level verification. In sequential microprocessor verification proofs, an assembly language instruction was verified in isolation. However, in our pipelined verification, the effects of control hazards created by branches can not be ignored.

For example, consider Figure 6.5. This diagram illustrates the effects of a taken branch, BEQ, on the SAWTOOTH Pipeline. Immediately after any conditional or unconditional branch instruction, a NOOP[1] instruction is inserted into the pipeline. This NOOP is important because it allows the designer to make sure that no state updating actions occur that can not be undone if the branch is taken. At this point, the instruction following the branch, SUB, is fetched. Now after this instruction fetch, the hardware

[1]In the diagram, the X's represent NOOP instructions.

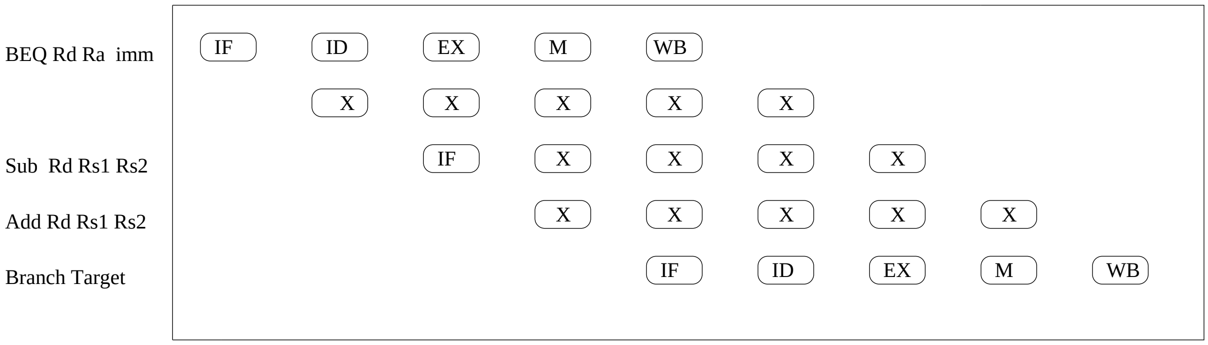

Figure 6.5: The Effects of a Taken Branch

can determine if the branch was taken because the condition code was updated in the third stage of the branch instruction. Since the branch is taken, the decoding mux `MX4C` replaces the SUB instruction with a NOOP instruction in the second stage of the SUB instruction. At the same time, the ADD instruction is not fetched but is replaced with a NOOP instruction. After the BEQ instruction completes its fourth stage, the branch target address is now known and the branch target can be fetched. The effects shown in Figure 6.5 also occur for the INT, RTI, and JALI instructions since they all act like taken branch instructions.

The effects illustrated here are a primitive form of branch prediction. The important thing to notice here is that the *architecture* used to handle control hazards in SAWTOOTH greatly affects our verification. For example, suppose we wished to verify the SUB or ADD instruction in Figure 6.5. For the ADD instruction, if `op3` is a taken branch then the ADD instruction was never meant to be there and verifying it at this point is meaningless.

For our verification of the top level instructions, we make an assumption that the instruction we wish to verify was meant to be. The `instr_exists_pred` definition, shown in Figure 6.3.2, states that for an instruction starting at time t:

- The instruction executing in stage four is not a taken branch (line 1).
- The instruction in stage four at time $t+1$ is not a taken branch (line 2).
- The instructions in stage four at time t and $t+1$ is not a INT, RTI or JALI (lines 3 and 4).
- The instruction immediately prior to the one we wish to verify is not an instruction that causes a NOOP to be inserted into the instruction stream (line 5).

In the verification of Uinta [WC94], the architecture used to handle control hazards also greatly affected the verification. In Uinta, the program counter is not updated until the third stage of an instruction. This *delay* gives the hardware enough time to determine if a branch is taken or not before updating the program counter. This means that a situation like the one in Figure 6.5 would not occur and the effects of branch instructions would not affect the verification of one instruction as they do here

```
instr_exists_pred =
⊢def ∀s t.
    instr_exists_pred s t =
     ¬((Op3P(s t)) IN {54,48,51,50,53,52,49,55} ∧ CondP(s t)) ∧
     ¬((Op3P(s(t + 1))) IN {54,48,51,50,53,52,49,55} ∧ CondP(s(t + 1))) ∧
     ¬(Op3P(s t)) IN {57,5,6} ∧
     ¬(Op3P(s(t + 1))) IN {57,5,6} ∧
     ¬(Op2P(s(t + 1))) IN {5,6,56,57,54,48,51,50,53,52,49,55} ∧
```

Figure 6.6: The Instruction Exists Predicate

in SAWTOOTH. However, in order to complete the verification, extra macro level state elements, **nextpc** and **nextnextpc** which do not really exist, had to be added to show that the program counter is updated correctly.

Another thing to notice about Figure 6.5 is that the program counter is not updated until time **t+4** for a taken branch. However for a branch that is not taken, the results are much different. For example consider the same situation as in Figure 6.5 but the branch is not taken. This scenario is shown in Figure 6.7. Here, the SUB instruction is allowed to continue in the decode stage since the hardware can detect that the branch was not taken. The ADD is then fetched normally and execution proceeds normally. In this diagram, the BEQ instruction really updates the program counter at time **t+1** since it is using a primitive form of branch prediction.

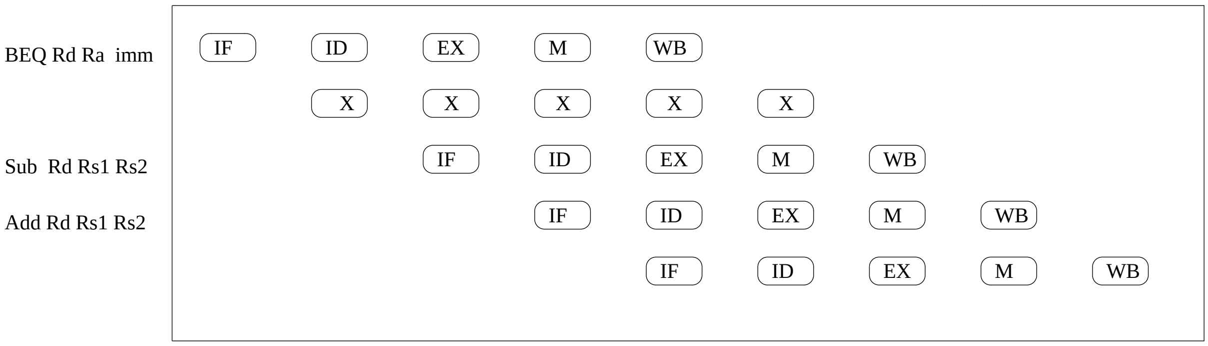

Figure 6.7: The Effects of a Branch Not Taken

6.3.3 Results

Noting the effects of the temporal abstraction described in the previous section, we are able to verify the architectural instructions. An example of the theorem we prove for each instruction is shown in Figure

6.8. This theorem states that if several assumptions are met then the state returned by the function `M_INST` for the ADD instruction is the same as the one computed by the pipeline interpreter (line 7 of the theorem). First, let's discuss the many assumptions in the theorem.

- **Line 1**. This assumption states that the next instruction we fetch from memory is an ADD.
- **Line 2**. This assumption is an unfortunate result of using the abstract representation. It states that if the instruction fetched is an ADD, then the word fetched can not be a 32-bit string of zeros. In other words, if the instruction fetched is an ADD, then it can not be a NOOP. Obviously, this assumption makes no sense for a NOOP instruction, however it makes no difference in the proof.
- **Line 3** This is the load assumption that was discussed in section 6.1.3.
- **Line 4** This is the assumption that the program does not have self-modifying code that was presented in section 6.1.1.
- **Line 5** These are the assumptions defined in section 4.2.14.
- **Line 6** This is the PPSW assumption that was defined in section 6.1.2.
- **line 8** This is the instruction exists predicate that we defined in the previous section.

  It is important to note that this is the *only* assumption we make about other instructions in the pipeline. No assumptions are made about other instructions in the pipeline that relate to data hazards or structural hazards caused by LOAD and STORE instructions.
- **line 9** This assumption says that we do not start in a state where the pipeline will be flushed.

In section 6.3, we discussed the effects of skewing, shifting, and stalling. Each one of these plays a part in the construction of this theorem.

Skew Skew occurs because the architectural state elements are updated at different time intervals while the instruction is in the pipeline. In the above theorem, for example, the register file is updated between time $t+4$ and time $t+5$ (lines 12 and 14). However, the program status word is updated between times $t+2$ and $t+3$ (lines 13 and 15).

Stalling The effect stalling has on the theorem in Figure 6.8 is shown on line 10. The existentially quantified variable `n` represents the number of stalls that occur during the execution of the instruction. Notice how `n` is added to the times of the states elements (line 13 for the register file). This is done to model the effects of the inserted NOOPs and ensure that the register file is updated at the correct time frame in our proof.

```
add_instr_lemma =
⊢ ∀decrep s' e' p' t x0 x1' x2''.
1                 (decode_word
                  decrep
                  (fetch mrep(MemP(s' t),address orep(Pc1P(s' t)))) =
                  ADD x0 x1' x2'') ⟹
2                ((decode_word
                   decrep
                   (fetch mrep(MemP(s' t),address orep(Pc1P(s' t)))) =
                   ADD x0 x1' x2'') ⟹
                  ¬(fetch mrep(MemP(s' t),address orep(Pc1P(s' t))) =
                    wordn orep 0)) ⟹
3                 load_assump orep decrep mrep s' ⟹
4                 memory_assump orep mrep ⟹
5                 phase_assums orep ⟹
6                 ppsw_assump orep decrep mrep s' ⟹
7                 Pipeline_Interp decrep orep mrep s' e' p' ⟹
8                 instr_exists_pred s' t ⟹
9                 ¬WipeP(s' t) ⟹
10                 (∃n.
11                   let j =
                     ((FST (Wordn_decode orep decrep (IrP(s'(t + (n + 1))))) = 56) →  1 |
                     (((FST(Wordn_decode orep decrep (IrP(s'(t +(n+1)))))) IN
                           {54,48,51,50,53,52,49,55} ∧ CondP(s'(t + (n + 3)))) →  3 |
                     ((FST(Wordn_decode orep decrep (IrP(s'(t + (n + 1)))))) IN {57,5,6} →  3 |  0)))
                   in
                    (M_INST orep mrep (ADD x0 x1' x2'')
                     (Macro_State
12                    (RegP(s'(t + 4)))
13                    (PswP(s'(t + 2)))
                      (Pc1P(s'(t + 0)))
                      (MemP(s'(t + 3)))
                      (Pa2P(s'(t + 1))))
                     (e' t) =
                     Macro_State
14                   (RegP(s'(t + (5 + n))))
15                   (PswP(s'(t + (3 + n))))
16                   (Pc1P(s'(t + (1 + (n + j)))))
                     (MemP(s'(t + (4 + n))))
                     (Pa2P(s'(t + (2 + n))))))
```

Figure 6.8: A verified ADD instruction

Shifting Shifting occurs because the program counter is sometimes updated at time intervals other than between times t and $t+1$. In our theorem the variable `j` (line 11) is used to model the effects of shifting. If the instruction we are verifying is a JUMP instruction then `j` is set 1. If the instruction we are verifying is a taken branch or an unconditional jump (INT, RTI, or JALI), then `j` is set to 3. Otherwise there is no shifting and `j` is set to 0. The value of `j` is added to line 15 to set the correct time for which the state update will take place for the program counter.

Overall, a theorem like the one in Figure 6.8 was proven for all of the 41 top level instructions. The importance of this theorem is that we *do not* make any assumptions about correct instruction pipelining— besides assuming the the instruction we wish to verify will not be flushed. The Data Forwarding Lemma, see section 6.2, was used to handle data hazards in the pipeline, and structural hazards due to LOAD and STORE instructions were dealt with using case analysis in the proof.

The proof of this theorem is very large. The subgoal tree for the **`add_instr_lemma`** is shown in Figure 6.9. The proof breaks down into four subgoals each representing the number of times our instruction is stalled. For each one if these subgoals, we must prove that the five state elements are each updated correctly. If we are verifying a branch instruction, the subgoal for the program counter breaks down into two more subgoals based on whether the branch is taken or not.

Proving the theorem in Figure 6.8 also lead directly to the discovery of a design error. While trying to prove that the register file is updated correctly in the case where there are no stalls, the PPSW error described in section 6.1.2 was discovered.

6.4 Flushed Instructions

In the previous section we presented the results of verifying instructions. However, each verified instruction was predicated on the fact that the instruction would not be flushed from the pipe. The predicate **`instr_exists_pred`** was used to make sure that the instruction we wished to verify would not be replaced by a NOOP instruction by the pipeline to prevent control hazards. In this section we examine what happens to an instruction when the **`instr_exists_pred`** is false.

The five conditions that the **`instr_exists_pred`**, shown in Figure 6.3.2, check for all insert a NOOP instruction into the pipeline. So at first glance, it would appear that if this predicate did not hold, then the net effect on the state would be the same as that of the NOOP instruction **`M_NOOP`** that is defined for the top level instructions. Indeed it is straightforward to prove that the register file, the program status word, memory, and the page address bit all remain unchanged during the time intervals most instructions would update them. The four theorems that state this are shown below.

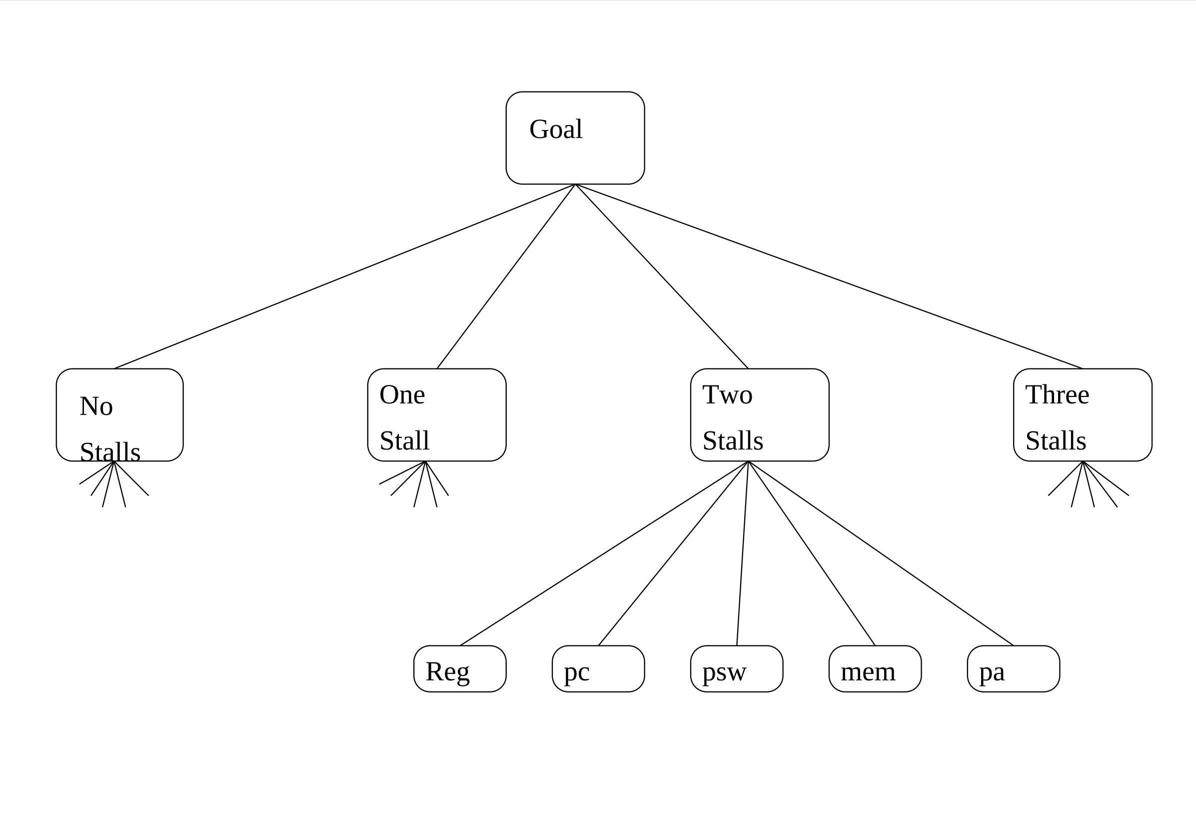

Figure 6.9: The Proof Tree for One Instruction

```
taken_branch_lemma6 =
⊢ ∀orep decrep mrep s' e' p'.
    phase_assums orep ⟹
    Pipeline_Interp decrep orep mrep s' e' p' ⟹
    ¬instr_exists_pred s' t ⟹
    (RegP(s'(((((t + 1) + 1) + 1) + 1) + 1)) =
     RegP(s'((((t + 1) + 1) + 1) + 1)))



taken_branch_lemma8 =
⊢ ∀orep decrep mrep s' e' p'.
    phase_assums orep ⟹
    Pipeline_Interp decrep orep mrep s' e' p' ⟹
    ¬instr_exists_pred s' t ⟹
    (MemP(s'((((t + 1) + 1) + 1) + 1)) = MemP(s'(((t + 1) + 1) + 1)))



taken_branch_lemma9 =
⊢ ∀orep decrep mrep s' e' p'.
    phase_assums orep ⟹
    Pipeline_Interp decrep orep mrep s' e' p' ⟹
    ¬instr_exists_pred s' t ⟹
    (PswP(s'(((t + 1) + 1) + 1)) = PswP(s'((t + 1) + 1)))



taken_branch_lemma10 =
⊢ ∀orep decrep mrep s' e' p'.
    phase_assums orep ⟹
    Pipeline_Interp decrep orep mrep s' e' p' ⟹
    ¬instr_exists_pred s' t ⟹
    (Pa2P(s'((t + 1) + 1)) = Pa2P(s'(t + 1)))
```

While the above lemmas were simple to prove, the updating of the program counter is a more interesting problem–a problem we did not solve. A NOOP instruction can be inserted into the pipeline at *two* locations: the mux that loads instruction register, **MXIR**, and the mux that decodes instructions **MX4C**. The VHDL and **HOL** specifications for these muxes are shown in sections 3.5 and 3.6.1 respectively. When a NOOP is inserted into the pipeline, the program counter can be updated in the following ways:

1. When the instruction in stage four of the pipeline is a taken branch or unconditional branch, then the program counter is updated with the value from the ALU. This value is the memory address for the branch target.

2. When the instruction being decoded in stage two of the processor is a JUMP instruction then the program is updated with the value of the 26-bit immediate in the JUMP instruction.

3. When the instruction in stage four is a LOAD or a STORE the program counter remains the same. Note that this condition is not in the predicate `instr_exists_pred`.

4. When the instruction being decoded in stage two is a branch instruction, then the program counter is once again left unchanged.

These state updates for `pc1` are specified in section 3.6.3. The important item to notice is that when a NOOP is inserted into the pipeline, the program counter can be updated with three different values. Two of these values, items 1 and 2 above, require pieces of state which are outside the architectural state for the `M_NOOP` instruction.

To date, we have not been able to prove an overall theorem about flushed instructions and inserted NOOPS because of these complications. Once again the important item to notice is that the specific architecture used for SAWTOOTH to deal with control hazards caused problems with the completion of this verification.

6.5 Interrupts

SAWTOOTH has support for interrupt instructions, INT, and external interrupts. An external interrupt occurs when the control line **iflush** is set high. When **iflush** is set to high, the mux that decodes instructions, **MX4C** shown in section 3.6.1, inserts the INT instruction into the pipeline. The VHDL for the **iflush** process is once again shown here.

```
do_iflush : PROCESS (exc, psw(3), wipe)
  BEGIN
  iflush <= exc AND (NOT psw(3)) AND (NOT wipe);
  END PROCESS do_iflush;
```

Notice that **iflush** will only be set high if the fourth bit of the program status word is not set high. The fourth bit of the **psw**, Figure 3.1, is the supervisory mode bit. This is an error. The designer should have checked to see if the interrupts are enabled and reference the interrupt enable bit instead. Unfortunately this error was discovered late in the design process.

Even though this design error was discovered, we still wanted to verify that the microprocessor design correctly implemented external interrupts. Before presenting the results of this effort, it is worth while to explain in some detail what happens when an external interrupt occurs.

As mentioned earlier, the **iflush** line signals the processor that an external interrupt has occurred. This signal is high when the exception line is high, the pipeline is not being wiped, and the processor is not in supervisory mode. When these conditions hold several things happen:

- The mux **MX4C** inserts an interrupt instruction into the pipeline *in stage two*. This is shown in section 3.6.1.
- The opcode registers **op3** and **op4** are updated with NOOP instructions. This effectively flushes the pipe. Only the instruction in stage five completes execution. Also the destination registers **rd3** and **rd4** are updated with the value of the **R0** register. This makes register write-backs to these registers NOOP instructions. The **HOL** and VHDL specifications for the opcode registers and destination registers are shown in sections 3.6.6 and 3.6.5 respectively.
- The **pc2** register is updated with the value in the **pc4** register. This is shown in section 3.6.3. This is important to notice. The **pc4** register holds the value which will be stored in the return register for the INT instruction. After the value is placed in **pc2**, it then is passed to **aluout1**, shown in section 3.6.2, and from there it is written to the register file. The important item to notice is that for our top level specification for external interrupts, there is no state element that corresponds to the value kept in **pc4**. That is, to fully understand the actions that occur in the case of interrupts, we must extend the state of the microprocessor at the architectural level.

To verify external interrupts for SAWTOOTH we created a new architectural state which contained the original five state elements plus an extra 32-bit word holding the value of return register **pc4**. With this new state, the definition **M_EINT** in section 5.3.2 is no longer adequate, so a new definition is presented below.

```
M_NEW_EINT =
⊢def ∀orep mrep s e.
    M_NEW_EINT orep mrep s e =
    (let reg = Regn s
     and psw = Pswn s
     and pc = Pcn s
     and mem = Memn s
     and pa2 = Pan s
     and rtn = Rtnn s  in
      let a = FETCH_REG orep reg R0 pa2 in
       let i = extend orep(low16w orep(Ivecp e)) in
        let result = band orep(add orep(a,i),wordn orep 255)  in
         let new_psw =
               ((shl orep(psw,8)) except
                [pa,get_pa orep psw;sm,T;ie,F;cf,get_cf orep psw;
                 vf,get_vf orep psw])
               orep   in
          let new_reg = MACRO_UPDATE_REG orep reg R1 pa2 psw rtn  in
           let new_pc = result  in
            let new_pa = get_pa orep psw  in
             let new_rtn = result in
              New_Macro_State new_reg new_psw new_pc mem new_pa new_rtn)
```

There are two differences between this definition and the one presented in the top level specifications:

- The register file is updated by writing to register `R1` the value in the return register–not the value in the program counter.
- The new state element, the return register, is updated with the target address for the jump just like the program counter. The reason for this is that the return register must follow the execution path of the program.

Now with this new specification for external interrupts, we prove the theorem shown in Figure 6.10. This theorem is much simpler than the theorem for a verified instruction. Once again let's start by explaining the assumptions of the theorem (by line number).

1. The assumption `R1_assump` is another unfortunate result of using the abstract representation. The assumption is shown in Figure 6.11.

   The assumption states the the fifth or high order bit of the register address `R1` is a 0. Now this is obvious since the 5-bit string for `R1` would be "00001", however we did not define any theory obligations about register addresses.

   The reason for this assumption can be seen in the specification for the destination register `rd2` and the mux that loads it, `MX2RD`, shown in section 3.6.5. One of the checks made is to see if the fifth bit of the register address is high. With the destination register being `R1`, we know this is false.

2. This assumption states that we do not receive a reset signal at any time.

3. This assumption states that any opcode we fetch or decode is valid. Note that assumptions 2 and 3 are two of the assumptions defined in section 4.2.14. They are broken apart here since we no longer assume that we do not get an exception.

```
INT_THM =
. ⊢ ∀decrep s′ e′ p′ t.
1      R1_assump orep ⟹
2      (∀e t1. ¬Resetp(e t1)) ⟹
3      (∀op.
        op IN
        32,33,34,8,9,10,11,12,13,14,16,17,18,19,20,21,22,23,24,25,26,
         27,28,29,1,2,3,4,5,6,0,56,57,48,49,50,51,52,53,54,55) ⟹
4      Pipeline_Interp decrep orep mrep s′ e′ p′ ⟹
5      ¬WipeP(s′ t) ⟹
6      Excp(e′ t) ⟹
7      ¬get_sm orep(PswP(s′ t)) ⟹
8      let rtn_shift = 2
9      in
       (M_NEW_EINT
        orep
        mrep
        (New_Macro_State
         (RegP(s′(t + 3)))
         (PswP(s′(t + 1)))
         (Pc1P(s′(t + 0)))
         (MemP(s′(t + 2)))
         (Pa2P(s′(t + 0)))
10       (Pc4P(s′(t + 0))))
        (e′ t) =
        New_Macro_State
        (RegP(s′(t + 4)))
        (PswP(s′(t + 2)))
        (Pc1P(s′(t + 3)))
        (MemP(s′(t + 3)))
        (Pa2P(s′(t + 1)))
11      (Pc4P(s′(t + (1 + rtn_shift)))))
Run time: 5.5s
```

Figure 6.10: The Interrupt Theorem

4. This assumption is the pipeline interpreter. We are trying to prove that the pipeline interpreter implements external interrupts correctly.

5. This assumption states the the pipeline is not being wiped at time t.

6. This assumption states that the exception flag from the environment is high.

7. This assumption states that the processor is not in supervisory mode. Note that assumptions 5-7 must hold to set the `iflush` line high and signal an external interrupt.

The theorem states that if we start in a state that causes an external interrupt to occur, then the state updated by the pipeline after 5 cycles is equal to the state updated by the function `M_NEW_EINT`.

The only item about the above theorem that is not similar to the verified instruction theorem is the temporal abstraction for the state element `rtn_reg`.

Since we need the value in the `pc4` register at time t to update `pc2` in the pipeline, we call the function `M_NEW_EINT` with the the return register mapped to the pipeline state element `pc4` at time t.

```
let R1_assump = new_definition
  (`R1_assump`,
  "! (orep:^OPS_type) .
     R1_assump orep = bit5 orep 5 R1 = F " );;
```

Figure 6.11: The `R1` Assumption

Line 11 of theorem states that we update the return register at time $t+1+\text{rtn\_shift}$. The shift involved here is similar to the shift involved with the program counter being updated at time $t+4$ for a taken branch instruction. Here, the return register, `pc4` is updated for the INT instruction between times $t+2$ and $t+3$. Because of this, the value of `rtn_shift` is added to line 11 of the theorem.

The important item to notice about this theorem is that the pipeline architecture used to flush the pipeline forced us to extend the top level state of the processor to fully model the actions that take place during an external interrupt. Since all of the instructions, except the one in stage five, are flushed for the interrupt, the INT instruction must save the value in the `pc4` register in order to restart these instructions after the interrupt is serviced. Since an equivalent of `pc4` was not in the architectural state, we had to add the state element `rtn_reg`.

6.6 Other Results

We have shown that the verification of SAWTOOTH has found two design errors in the processor. We also found a couple of design errors by just a careful analysis of the VHDL–which is necessary to write the formal specifications. The two errors concern supervisory registers and interrupts. The error concerning the VHDL for the process `iflush` was discussed in the previous section.

Supervisory Registers In section 3.6.5 we defined the behavior of the `MX2RD` multiplexor. This mux outputs the value of the destination register for an instruction and the result is stored in the `rd2` register. Early attempts at writing a pipeline specification turned up an error in the VHDL for this process. In the logic for this process, the mux outputs the destination register `R0` if the destination register from the instruction is a supervisory mode register and the processor is not in supervisory mode. We discovered in early designs that the processor did not do this correctly and situations existed in which a write to a supervisory mode register by a process not in supervisory mode could take place. Fortunately, this error was discovered early and has been corrected.

6.7 Summary

In this chapter, we have shown the results of our verification of the architectural level instructions, the verification of external interrupts, and a discussion of effects of inserted NOOP instructions. We also presented the necessary assumptions for these results to be valid. Two of these assumptions were created to deal with design errors in the processor.

Chapter 7

Conclusions and Further Research

In thesis, we presented the results of the verification of the SAWTOOTH microprocessor. We presented the specification of the processor from the electronic block model to the architectural specifications. We showed that the electronic block model correctly implemented the phase level specifications and that the phase level correctly implemented the pipeline level. We then proved that the pipeline level specifications implemented each of the 41 top level instructions. We also presented a correctness theorem for external interrupts, and we presented a discussion on the effects of inserted NOOP instructions. Although we were not able to to prove the final theorem stating that the sequential machine (the architectural level) behaves in the same manner as the parallel machine (the pipeline level), we believe our investigations here have presented some insight into some benefits and some future research topics in the area of microprocessor verification.

Benefits to Design One of the major goals of this exercise was to show that hardware verification can be a useful tool for industry. In this verification we found several design errors as a direct result of the specification and verification. Three of these errors, the PPSW error shown in section 6.1.2, the LOAD error shown in section 6.1.3, and the INT error shown in section 6.6, were found even though the circuit had been simulation tested by the designer.

The LOAD error and PPSW error were discovered late in the design process so software assumptions were created to finish the verification. These software assumptions highlight two points. Using these two assumptions we can now create a scanner that could check assembly code to see if it will invalidate the proof. This scanner would aid in the design of a compiler for SAWTOOTH, which could be guaranteed to produce code that would not invalidate the verification efforts. Secondly, the verification effort for this processor started when the designer was well into the design of the processor. We believe that a closer working relationship between the designer and verifier from the start could have led to the discovery of these errors at an early enough time to correct them. Indeed, as shown in section 6.5 and section 6.6, the writing of the formal specifications alone often will highlight design errors.

Future Research The verification research in this thesis is an extension of the work done by Windley and Coe[WC94]. The major difference between SAWTOOTH and Uinta was how each processor handled structural hazards caused by conditional and unconditional jump instructions as discussed in section 6.3. The importance of their difference is their impact on the verifications. For Uinta, the branch delay prevented instructions from entering the pipe that were not the proper branch target. However in SAWTOOTH the primitive branch prediction allowed instructions into the pipe even though the may not have been the proper branch target. For the verification we made an assumption that stated, for each instruction we verified, the instruction was not one that would get flushed out of the pipeline.

The effects of using this primitive branch prediction on the verification were highlighted even more when we considered what happens when instructions do get flushed from the pipeline in section 6.4. Because of the architecture used, our model could not specify entirely how the program counter was updated using a traditional architectural state.

A similar problem was found when verifying the correct behavior of external interrupts in section 6.5. To correctly model the behavior of external interrupts in the pipeline, an additional state element, the return register, had to be added to the top level state. Once again, the specific architecture used by the designer for interrupts caused this modification to our verification efforts.

Based on these results, two items become clear:

- If one wishes to build a pipelined microprocessor and verify it, the processor must be designed from the start with verification in mind. Certainly, with SAWTOOTH, an alternative architectural feature to deal with control hazards could have been used to ease the complications of the verification–namely the verification of flushed instructions which were allowed to enter the pipeline. This point also reinforces a point made earlier. If the designer and verification specialist have a much closer working relationship, problems like these could be avoided.

- We have stated before [WC94, page 16] that using the experience from the Uinta proof and this verification, we intended to modify the generic interpreter theory to include pipelined architectures. Based on the experience of this effort, we believe this task may be much more difficult than imagined or nearly impossible. Since each pipeline architecture uses special design features to handle hazards–especially control hazards–generalizing this behavior for the generic interpreter theory seems a difficult task that will require much more research.

Even though this verification effort did not conclude with an overall correctness like the one presented for Uinta[WC94, page 15], we believe that work presented here to an important basis for continued research into the area of microprocessor verification.

Bibliography

[ALL$^+$93] Tej Arora, Tony Leung, Karl Levitt, et al. Report on the UCD Microcoded Viper Verification project. In *Proceedings of the HOL User's Group Workshop*, pages 241–255, 1993.

[Bar84] H. G. Barrow. Verify: A program for proving correctness of digital hardware designs. *Artificial Intelligence*, 24, 1984.

[BG90] G. Birtwistle and B. Graham. Verifying SECD in HOL. In J. Staunstrup, editor, *Formal Methods for VLSI Design*. Elsevier Science Publishers, 1990.

[BM79] Robert S. Boyer and J. Strother Moore. *ACM Monograph Series: A Computational Logic*. Academic Press Inc., 1979.

[BMS89] M. Bickford, C. Mills, and E.A. Schneider. Clio: An applicative language-based verification system. Technical Report TR 15-7, Osyssey Research Associates, Inc., 1989.

[CGM87] Albert Camelleri, Mike Gordon, and Tom Melham. Hardware verification using higer order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Circuit Designs*. Elsevier Scientific Publishers, 1987.

[Chu40] A. Church. A formulation of the simpole theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Coe89] David R. Coello. *The VHDL Handbook*. Kluwer Academic Publishers, Boston, 1989.

[Coh88a] Avra Cohn. Correctness properties of the viper block model: The second level. Technical Report 134, University of Cambridge Computer Laboratory, 1988.

[Coh88b] Avra Cohn. A proof of correctness of the viper microporocessor: the first level. In G. Birtwhistle an P. Subrahmanyam, editor, *VLSI Specification, Verification, and Synthesis*. Kluwer Academic Publishers, 1988.

[CP88] Paolo Camurati and Paolo Prinetto. Formal verification of hardware correctness: Introduction and survey of current research. *IEEE Computer*, pages 8–19, July 1988.

[Cul88] W. J. Cullyer. Implementing safety-critical systems: The VIPER microprocessor. In G. Birtwhistle and P. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*. Kluwer Academic Publishers, 1988.

[CW92] Michael L. Coe and Phillip J. Windley. Microprocessor verification-a tuturial. Technical Report LAL 92-10, University of Idaho, Labratory for Applied Logic, 1992.

[Gor83] M. Gordon. Proving a computer correct. Technical Report 42, University of Cambridge, 1983.

[Gor88] Michael J.C. Gordon. HOL: a proof generating system for higher-order-logic. In *VLSI Specification, Verification, and Synthesis*. Kluwer Academic Press, 1988.

[Hen80] P. Henderson. *Functional Programming; Applications and Implementation*. Prentice-Hall, London, 1980.

[HP90] John L. Hennessy and David A. Patterson. *Computer Acrhitecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.

[Hun87] W. A. Hunt. The mechanical verification of a microprocessor design. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*. North-Holland, 1987.

[Hun89] Warren A. Hunt. Microprocessor design verification. *Journal of Automated Reasoning*, 5:429–460, 1989.

[Joy88] Jeffry Joyce. Formal verification amd implementation of a microprocessor. In G. Birtwhistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*. Kluwer Academic Press, 1988.

[Joy89] Jeffrey Joyce. *Multi-Level Verification of Microprocessor-Based Systems*. PhD thesis, Cambridge University, 1989.

[KN85] D. Kapur and P. Narendran. An equational apporoach to theorem proving in first-order predicate calculus. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, 1985.

[Lan64] P.J. Landin. The mechanical evalutation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[Lee92] M. E. Leeser. Using Nuprl for the verification and synthesis of hardware. In C.A.R. Hoare and M.J.C. Gordon, editors, *Mechanized Reasoning and Hardware Design*. Prentice Hall, 1992.

[LSU89] Roger Lipsett, Carl Schaefer, and Cary Ussery. *VHDL: Hardware Description and Design*. Kluwer Academic Publishers, Boston, 1989.

[Mel88] Thomas Melham. Abstaction mechanisms for hardware verification. In G. Birtwhistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*. Kluwer Academic Publishers, 1988.

[NS89] Paliath Narendran and Jonathan Stillman. Formal verification of the sobel image processing chip. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer-Verlag, 1989.

[SB90a] Mandayam Srivas and Mark Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, 7(5):52–64, September 1990.

[SB90b] Mandayam Srivas and Mark Bickford. Verification of a pipelined microprocessor using clio. In *Hardware Specification, Verification, and Synthesis: Mathematical Aspects*. Springer-Verlag, 1990.

[SD88] Computer Science and Electrical Eng. Depts. Cayuga chip specification release 2.2. Technical report, Cornell University, 1988.

[Soc88] IEEE Computer Society. *IEEE Standard VHDL Language Reference Manual*. Institute of Electrical and Electronics Engineers, New York, 1988.

[TK93] Sofiéne Tahar and Ramayya Kumar. Implementing a methodology for formally verifying RISC processors in HOL. In *Proceedings of the HOL User's Group Workshp*, pages 283–296, 1993.

[WC94] Phillip J. Windley and Michael L. Coe. A correctness model for pipelined microprocessors. Technical report, Laboratory for Applied Logic, Brigham Young University, 1994.

[Win90a] Phillip J. Windley. *Formal Verification of Generic Interpreters*. PhD thesis, University of California, Davis, 1990.

[Win90b] Phillip J. Windley. The verification of AVM-1. Technical Report CSE 90-21, University of California, Davis, 1990.

[Win92] Phillip J. Windley. Abstract theories in HOL. In *Proceedings of the International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 17–30, 1992.

[Win93] Phillip J. Windley. A theory of generic interpreters. In George J. Milne and Laurence Pierre, editors, *Correct Hardware Design and Verification Methods*, number 683 in Lecture Notes in Computer Science, pages 122–134. Springer-Verlag, 1993.

[Win94] Phillip J. Windley. Architectural specification in HOL. Techinical report, Laboratory for Applied Logic, Brigham Young University, 1994.

Appendix A

The Abstract Representation

Table A.1 shows the full abstract representation that was used to model operations on bit strings.

Using the abstract functions defined in A.1 we can model the microprocessor without having to worry too much about the very low level implementation details. Unfortunately, since these functions are abstract we don't know anything about there behavior. Using the abstract theory package[Win92] we define a series of small assumptions that will be neccessary for the verification to be valid. These assumptions are created by the **HOL** session below.

```
new_theory_obligations
   (`Ops_thobs`,
    "! rep:^OPS_type . Ops_thobs rep =
     (! a (b:*wordn) . add rep (a,b) = (add rep (b,a))) /\
     (! a (b:*wordn) . umult rep (a,b) = (umult rep (b,a))) /\
     (! a (b:*wordn) . mult rep (a,b) = (mult rep (b,a))) /\
     (! a (b:*wordn) . band rep (a,b) = (band rep (b,a))) /\
     (! a (b:*wordn) . bxor rep (a,b) = (bxor rep (b,a))) /\
     (! a (b:*wordn) . bor rep (a,b) = (bor rep (b,a))) /\
     (! a (b:*wordn) . eqp rep (a,b) = (eqp rep (b,a))) /\
     (! n m w1 b .
      ~(n = m) ==> (bit rep m(setbit rep(n,w1,b)) = bit rep m w1)) /\
     (! m w1 b . (bit rep m(setbit rep(m,w1,b)) = b)) /\
     (! n m w1 b1 b2 .
      ~(n = m) ==> (setbit rep (m,(setbit rep(n,w1,b1),b2)) =
                    setbit rep (n,(setbit rep(m,w1,b2),b1)))) /\
     (! b n w1 . setrange rep w1 (n,n) (repeat rep (1,b)) =
                 setbit rep (n,w1,b))"
   );;
```

Function	Type	Description
add	:*wordn # *wordn -> *wordn	Adds two n-bit words
addc	:*wordn # *wordn # bool -> *wordn	Add with carry
sub	:*wordn # *wordn -> *wordn	Subtract two n-bit words
subc	:*wordn # *wordn # bool -> *wordn	Subtract with carry
band	:*wordn # *wordn -> *wordn	Bitwise And
bxor	:*wordn # *wordn -> *wordn	Bitwise Xor
bor	:*wordn # *wordn -> *wordn	Bitwise Or
bnot	:*wordn -> *wordn	Bitwise Not
mult	:*wordn # *wordn -> *wordn # *wordn	Multiplication
umult	:*wordn # *wordn -> *wordn # *wordn	Unsigned multiplication
div	:*wordn# *wordn -> *wordn # *wordn	Division
udiv	:*wordn # *wordn -> *wordn # *wordn	Unsigned division
eqp	:*wordn # *wordn -> bool	Equality
gtzp	:*wordn -> bool	Greater than zero
gtezp	:*wordn ->bool	Greater than or equal to zero
ltzp	:*wordn -> bool	Less than zero
ltezp	: *wordn -> bool	Less than or equal to zero
shl	:*wordn # num -> *wordn	Shift left
shr	:*wordn # num -> *wordn	Shift right
bit	:num -> *wordn -> bool	Get a bit from a word
bit5	:num -> *word5 -> bool	Get a bit from a word5
bit6	:num -> *word6 -> bool	Get a bit from a word6
setbit	:num # *wordn # bool -> *wordn	Set a bit in a word
setb6	:num # *word6 # bool -> *word6	Set a bit in a word6
val	:*wordn -> num	Numeric value of a word
val6	:*word6 -> num	Numeric value of word6
wordn	:num -> *wordn	Wordn value of a number
address	:*wordn -> *address	Address representation of a number
subrange	:*wordn -> (num # num) -> *wordn	Subrange a word from n to m
repeat	:num # bool -> *wordn	Repeat a bit n times
setrange	:*wordn -> (num # num) -> *wordn -> *wordn	Set a range in a word
extend	:*short -> *wordn	Extend immediate value to word
extend5	:*word5 -> *word6	Extend word5 to word6
low16w	:*wordn -> *short	Take lower 16 bits of a word
extend26	:*word26 -> *wordn	Extend word26 to wordn
extendb	:*byte-> *short	Extend a byte to half-word

Table A.1: The Abstract Representation for Bit Vectors

Appendix B

The Assembly Language Instructions

In this appendix, we present tables displaying information about each instruciton. There are six tables–one for each type of instruction. Each table presents: the opcode for each instrution, the opcode format as shown in figure 3.2, the two ALU inputs for the instruciton, and the ALU action for that instruction.

Table B.1: Load/Store Operations

OPCODE	Number	Form.	Description	ALU1	ALU2	Funct.
LD	40	2	Load a register from memory	Ra	ImmA	+
LHI	41	2	Load Hi using an immediate value	Ra	ImmA	LHI
ST	42	3	Store a register to memory	Ra	ImmA	+

Table B.2: Arithmetic Immediate Instructions

OPCODE	Number	Form.	Description	ALU1	ALU2	Funct.
ADDI	10	2	Add with immediate	Ra	Imm	+
ADDCI	11	2	Add with immediate using carry	Ra	Imm	+
SUBI	12	2	Subtract with immediate	Ra	Imm	-
SUBCI	13	2	Subtract with immediate using borrow	Ra	Imm	-
BANDI	14	2	Bitwise AND using immediate	Ra	Imm	AND
BORI	15	2	Bitwise OR using immediate	Ra	Imm	OR
BXORI	16	2	Bitwise XOR using immediate	Ra	Imm	XOR

Table B.3: Arithmetic Instructions

OPCODE	Number	Form.	Description	ALU1	ALU2	Funct.
ADD	20	1	Add	Ra	Rb	+
ADDC	21	1	Add with carry	Ra	Rb	+
SUB	22	1	Subtract	Ra	Rb	-
SUBC	23	1	Subtract with borrow	Ra	Rb	-
BAND	24	1	Bitwise AND	Ra	Rb	AND
BOR	25	1	Bitwise OR	Ra	Rb	OR
BXOR	26	1	Bitwise XOR	Ra	Rb	XOR
BNOT	27	1	Bitwise NOT	Ra	Rb	NOT

Table B.4: Shift Instructions

OPCODE	Number	Form.	Description	ALU1	ALU2	Funct.
LSL	30	1	Logical shift left	Ra		SL
LSR	31	1	Logical shift right	Ra		SR
ASR	32	1	Arithmetic shift right	Ra		SR
BSL	33	1	Byte shift left	Ra		BSL
BSR	34	1	Byte shift right	Ra		BSR
ABSR	35	1	Arithmetic byte shift right	Ra		BSR

Table B.5: Special Instructions

OPCODE	Number	Form.	Description	ALU1	ALU2	Funct.
NOP	00	4	No operation			
SPA	01	4	Set page address	Ra	Imm	SPA
RPA	02	2	Return page address			RPA
GPSW	03	2	Get program status word			GPSW
PPSW	04	2	Put program status word	Ra	Imm	PPSW
INT	05	2	User interrupt	Ra	Imm	INT
RTI	06	2	Return from interrupt	Ra	Imm	+

Table B.6: Jump and Branch Instructions

OPCODE	Number	Form.	Description	ALU1	ALU2	Funct.
JMP	70	4	Jump			
JALI	71	2	Jump and link with immediate	Ra	ImmA	+
BEQ	60	3	Branch if equal	PC1	ImmA	+
BNEQ	61	3	Branch if not equal	PC1	ImmA	+
BGTZ	62	3	Branch if greater than zero	PC1	ImmA	+
BGTEZ	63	3	Branch if greater than or equal to zero	PC1	ImmA	+
BLTZ	64	3	Branch if less than zero	PC1	ImmA	+
BLTEZ	65	3	Branch if less than or equal to zero	PC1	ImmA	+
BC	66	3	Branch on carry	PC1	ImmA	+
BOV	67	3	Branch on overflow	PC1	ImmA	+

Appendix C

HOL Definitions

In this appendix we present some of the HOL definitions we used frequently in the specification.

C.1 The Register File

The function UPDATE_REG is used to model the register write back in stage five of the pipeline. The function takes as input the register index, n, a register file, and the new value to written to n.

As stated earlier register R0 is reserved to be read-only and holds the value zero. Therefore if the register we are writing to is R0 then the register file is not updated. Otherwise register n is altered to the new value.

```
UPDATE_REG=
⊢def ∀n reg value orep.
    UPDATE_REG orep n reg value = ((n = R0) →  reg | ALTER reg n value)
```

The function FETCH_REG is used in the specification of the top level instructions. The definition of FETCH_REG hides the action of appending the page address bit the register index before fetching the register value. This function was used to make the macro level instruction defintions more readable.

```
FETCH_REG =
⊢def ∀orep reg Ra pa2.
    FETCH_REG orep reg Ra pa2 =
    (let source =
           ((¬pa2) →
            setb6 orep(5,extend5 orep Ra,pa2) |
            setb6 orep(5,setb6 orep(4,extend5 orep Ra,F),bit5 orep 5 Ra))
     in
      reg source)
```

The function `MACRO_UPDATE_REG` is used to specify the updating of the register file at the top level. Once again, it is used mainly to hide the details of the page address bit from the instruction definitions.

```
MACRO_UPDATE_REG =
⊢def ∀orep reg Rd pa2 value psw.
    MACRO_UPDATE_REG orep reg Rd pa2 psw value =
    (let dest =
           ((get_sm orep psw ∨ ¬pa2 ∨ ¬bit5 orep 5 Rd) →
            ((¬pa2) →
             setb6 orep(5,extend5 orep Rd,pa2) |
             setb6
             orep
             (5,setb6 orep(4,extend5 orep Rd,F),bit5 orep 5 Rd)) |
            R0)
     in
      UPDATE_REG orep dest reg value)
```

The function `INST_Decode` is used to transform an instruction of type `Inst_type` to a tuple of values representing the fields of an instruction opcode.

C.2 Decoding Instructions

```
INST_Decode=
⊢def (∀Rd Ra imm. INST_Decode(LD Rd Ra imm) = 32,Rd,Ra,ARB,imm,ARB) ∧
   (∀Rd Ra imm. INST_Decode(LHI Rd Ra imm) = 33,Rd,Ra,ARB,imm,ARB) ∧
   (∀Rd Ra imm. INST_Decode(ST Rd Ra imm) = 34,Rd,Ra,ARB,imm,ARB) ∧
   (∀Rd Ra imm. INST_Decode(ADDI Rd Ra imm) = 8,Rd,Ra,ARB,imm,ARB) ∧
   (∀Rd Ra imm. INST_Decode(ADDCI Rd Ra imm) = 9,Rd,Ra,ARB,imm,ARB) ∧
   (∀Rd Ra imm. INST_Decode(SUBI Rd Ra imm) = 10,Rd,Ra,ARB,imm,ARB) ∧
   (∀Rd Ra imm. INST_Decode(SUBCI Rd Ra imm) = 11,Rd,Ra,ARB,imm,ARB) ∧
   (∀Rd Ra imm. INST_Decode(BANDI Rd Ra imm) = 12,Rd,Ra,ARB,imm,ARB) ∧
   (∀Rd Ra imm. INST_Decode(BORI Rd Ra imm) = 13,Rd,Ra,ARB,imm,ARB) ∧
   (∀Rd Ra imm. INST_Decode(BXORI Rd Ra imm) = 14,Rd,Ra,ARB,imm,ARB) ∧
   (∀Rd Ra Rb. INST_Decode(ADD Rd Ra Rb) = 16,Rd,Ra,Rb,ARB,ARB) ∧
   (∀Rd Ra Rb. INST_Decode(ADDC Rd Ra Rb) = 17,Rd,Ra,Rb,ARB,ARB) ∧
   (∀Rd Ra Rb. INST_Decode(SUB Rd Ra Rb) = 18,Rd,Ra,Rb,ARB,ARB) ∧
   (∀Rd Ra Rb. INST_Decode(SUBC Rd Ra Rb) = 19,Rd,Ra,Rb,ARB,ARB) ∧
   (∀Rd Ra Rb. INST_Decode(BAND Rd Ra Rb) = 20,Rd,Ra,Rb,ARB,ARB) ∧
   (∀Rd Ra Rb. INST_Decode(BOR Rd Ra Rb) = 21,Rd,Ra,Rb,ARB,ARB) ∧
   (∀Rd Ra Rb. INST_Decode(BXOR Rd Ra Rb) = 22,Rd,Ra,Rb,ARB,ARB) ∧
   (∀Rd Ra. INST_Decode(BNOT Rd Ra) = 23,Rd,Ra,ARB,ARB,ARB) ∧
   (∀Rd Ra. INST_Decode(LSL Rd Ra) = 24,Rd,Ra,ARB,ARB,ARB) ∧
   (∀Rd Ra. INST_Decode(LSR Rd Ra) = 25,Rd,Ra,ARB,ARB,ARB) ∧
   (∀Rd Ra. INST_Decode(ASR Rd Ra) = 26,Rd,Ra,ARB,ARB,ARB) ∧
   (∀Rd Ra. INST_Decode(BSL Rd Ra) = 27,Rd,Ra,ARB,ARB,ARB) ∧
   (∀Rd Ra. INST_Decode(BSR Rd Ra) = 28,Rd,Ra,ARB,ARB,ARB) ∧
   (∀Rd Ra. INST_Decode(ABSR Rd Ra) = 29,Rd,Ra,ARB,ARB,ARB) ∧
   (∀imm. INST_Decode(SPA imm) = 1,ARB,ARB,ARB,imm,ARB) ∧
   (∀Rd. INST_Decode(RPA Rd) = 2,Rd,ARB,ARB,ARB,ARB) ∧
   (∀Rd. INST_Decode(GPSW Rd) = 3,Rd,ARB,ARB,ARB,ARB) ∧
   (∀Rd. INST_Decode(PPSW Rd) = 4,Rd,ARB,ARB,ARB,ARB) ∧
   (∀Rd Ra imm. INST_Decode(INT Rd Ra imm) = 5,Rd,Ra,ARB,imm,ARB) ∧
   (∀Rd imm. INST_Decode(RTI Rd imm) = 6,Rd,ARB,ARB,imm,ARB) ∧
   (INST_Decode NOOP = 0,ARB,ARB,ARB,ARB,ARB) ∧
   (∀imm26. INST_Decode(JMP imm26) = 56,ARB,ARB,ARB,ARB,imm26) ∧
   (∀Rd Ra imm. INST_Decode(JALI Rd Ra imm) = 57,Rd,Ra,ARB,imm,ARB) ∧
   (∀Rd Ra imm. INST_Decode(BEQ Rd Ra imm) = 48,Rd,Ra,ARB,imm,ARB) ∧
   (∀Rd Ra imm. INST_Decode(BNEQ Rd Ra imm) = 49,Rd,Ra,ARB,imm,ARB) ∧
   (∀Rd imm. INST_Decode(BGTZ Rd imm) = 50,Rd,ARB,ARB,imm,ARB) ∧
   (∀Rd imm. INST_Decode(BGTEZ Rd imm) = 51,Rd,ARB,ARB,imm,ARB) ∧
   (∀Rd imm. INST_Decode(BLTZ Rd imm) = 52,Rd,ARB,ARB,imm,ARB) ∧
   (∀Rd imm. INST_Decode(BLTEZ Rd imm) = 53,Rd,ARB,ARB,imm,ARB) ∧
   (∀imm. INST_Decode(BC imm) = 54,ARB,ARB,ARB,imm,ARB) ∧
   (∀imm. INST_Decode(BOV imm) = 55,ARB,ARB,ARB,imm,ARB)
```

The function `Wordn_decode` is used to transform a value of type `*wordn` to a tuple of values representing the fields of an instruction opcode. This function calls `INST_Decode` defined above. `Wordn_decode` also uses the function `decode_word`. `decode_word` is an abstract function that transforms objects of type `*wordn` to `Inst_type`.

The reader should note the comparison made by this function to check if the input equals a 32-bit word of all zeros. To model inserting a NOOP instruction into the pipeline, we use the notation `wordn orep 0` which means a 32-bit string of zeros. However, since both `wordn` and `decode_word` are abstract functions, we have no connection between an expression like `wordn orep 0` and the output of `decode_word`. Therefore we make this check to ensure we get the correct output from `decode_word`.

```
Wordn_decode=
⊢def ∀decrep orep w.
    Wordn_decode orep decrep w =
    ((w = wordn orep 0) →
     INST_Decode NOOP |
     INST_Decode(decode_word decrep w))
```

C.3 Miscellaneous Functions

Throughout, the **HOL** specifications several smaller functions were used that were built on top of these abstract functions.

Of these, the most obscure is the function `except`. `except` is a infix binary operator. The two operands this function requires are a 32-bit word and a list. Each element of the list is a pair where the first element of pair designates a certain bit in a word and the second element is a boolean value. An expression of the form:

```
w except [ (2,F)]
```

results in a new word with the value of the second bit set to `false`.

Most of these smaller functions were written to increase the readability of the specifications. Shown below are several of these small functions. Most of them concern the program status word.

$\vdash_{def}$ `msbv = 31`

$\vdash_{def}$ `∀orep. msb orep = bit orep msbv`

$\vdash_{def}$ `lsbv = 0`

$\vdash_{def}$ `∀orep. lsb orep = bit orep lsbv`

$\vdash_{def}$ `bit1 = 1`

$\vdash_{def}$ `∀orep w1. inc orep w1 = add orep(w1,wordn orep 4)`

$\vdash_{def}$ `∀orep w1. lower16 orep w1 = subrange orep w1(15,0)`

$\vdash_{def}$ `∀orep w1 w2. combine orep(w1,w2) = setrange orep w2(31,16)w1`

$\vdash_{def}$ `vf = 0`

$\vdash_{def}$ `∀orep psw. get_vf orep psw = bit orep vf psw`

$\vdash_{def}$ `cf = 1`

$\vdash_{def}$ `∀orep psw. get_cf orep psw = bit orep cf psw`

$\vdash_{def}$ `ie = 2`

$\vdash_{def}$ `∀orep psw. get_ie orep psw = bit orep ie psw`

$\vdash_{def}$ `sm = 3`

$\vdash_{def}$ `∀orep psw. get_sm orep psw = bit orep sm psw`

`pa =` $\vdash_{def}$ `pa = 4`

$\vdash_{def}$ `∀orep psw. get_pa orep psw = bit orep pa psw`

Appendix D

The Architectural Level Instructions

In this appendix we present the rest of the macro level instruction defintions.

```
M_LD =
⊢def ∀orep mrep Rd Ra imm s e.
    M_LD orep mrep Rd Ra imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s
     in
     let a = FETCH_REG orep reg Ra pa2
          and i = signextend orep imm  in
     let result = fetch mrep(mem,address orep(add orep(a,i)))  in
     let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
        and new_pc = inc orep pc  in
     let new_pa2 = pa2 in Macro_State new_reg psw new_pc mem new_pa2)
```

The LOAD instruction loads a word from memory. The effective address for the load is calculated by adding the value in register `Ra` with the immediate value.

```
M_LHI =
⊢def ∀orep mrep Rd Ra imm s e.
    M_LHI orep mrep Rd Ra imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s
     in
      let a = FETCH_REG orep reg Ra pa2  in
       let rs2 = signextend orep imm  in
        let result = combine orep(lower16 orep rs2,lower16 orep a) in
         let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result  in
          let new_pc = inc orep pc in
           Macro_State new_reg psw new_pc mem pa2)
```

The LHI instruction loads the uppper 16 bits of register `Rd` with the 16-bit immediate value and the lower 16 bits of register `Rd` with the lower 16 bits of the contents of register `Ra`.

```
M_ST =
⊢def ∀orep mrep Rd Ra imm s e.
    M_ST orep mrep Rd Ra imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s
     in
      let i = signextend orep imm in
      let rs1 =
             ((¬pa2) →
              setb6 orep(5,extend5 orep Ra,pa2) |
              setb6
              orep
              (5,setb6 orep(4,extend5 orep Ra,F),bit5 orep 5 Ra))  in
      let new_address = address orep(add orep(reg rs1,i))   in
      let new_pc = inc orep pc in
      let rd =
                ((get_sm orep psw ∨ ¬pa2 ∨ ¬bit5 orep 5 Rd) →
                 ((¬pa2) →
                  setb6 orep(5,extend5 orep Rd,pa2) |
                  setb6
                  orep
                  (5,setb6 orep(4,extend5 orep Rd,F),bit5 orep 5 Rd)) |
                 R0)  in
           let new_mem = store mrep(mem,new_address,reg rd)   in
            Macro_State reg psw new_pc new_mem pa2)
```

The STORE instruction stores the contents of the register `Rd` to memory at the location determined by adding the contents of register `Ra` and the immediate value. Note the functions FETCH_REG and MACRO_UPDATE_REG are not used here becuase of the slightly diffent opcode format of the STORE instruction and the instruction does not update the register file. Therefore the definition looks rather cluttered with the page address bit notations includeded.

```
M_ADDI =
⊢def ∀orep mrep Rd Ra imm s e.
    M_ADDI orep mrep Rd Ra imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s
     in
     let a = FETCH_REG orep reg Ra pa2
         and i = extend orep imm  in
     let result = add orep(a,i)   in
     let cflag = addp orep(a,i)
        and vflag = aovfl orep(a,i)   in
     let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
         and new_psw = (psw except [vf,vflag;cf,cflag])orep
         and new_pc = inc orep pc  in
          Macro_State new_reg new_psw new_pc mem pa2)
```

The ADDI instruction stores into the `Rd` register the sum of the value in the `Ra` register and the immediate value.

```
M_ADDCI =
⊢def ∀orep mrep Rd Ra imm s e.
    M_ADDCI orep mrep Rd Ra imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s
     in
     let a = FETCH_REG orep reg Ra pa2
      and i = extend orep imm
      and carry = get_cf orep psw  in
      let result = addc orep(a,i,carry)  in
      let cflag = addcp orep(a,i,carry)
        and vflag = acovfl orep(a,i,carry)  in
      let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
         and new_psw = (psw except [vf,vflag;cf,cflag])orep
         and new_pc = inc orep pc  in
          Macro_State new_reg new_psw new_pc mem pa2)
```

The ADDCI instruction stores the sum of the contents of the `Ra` register and the immediate value into the `Rd` register. The add is performed including the carry bit.

```
M_SUBI =
⊢def ∀orep mrep Rd Ra imm s e.
    M_SUBI orep mrep Rd Ra imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s
     in
     let a = FETCH_REG orep reg Ra pa2
      and i = extend orep imm  in
     let result = sub orep(a,i)   in
     let cflag = subp orep(a,i)
        and vflag = sovfl orep(a,i)  in
     let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
         and new_psw = (psw except [vf,vflag;cf,cflag])orep
         and new_pc = inc orep pc  in
          Macro_State new_reg new_psw new_pc mem pa2)
```

The SUBI instruction stores into the `Rd` register the difference between the contents of the `Ra` register and the immediate.

```
M_SUBCI =
⊢def ∀orep mrep Rd Ra imm s e.
    M_SUBCI orep mrep Rd Ra imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s
     in
     let a = FETCH_REG orep reg Ra pa2
      and i = extend orep imm
      and carry = get_cf orep psw  in
       let result = subc orep(a,i,carry)  in
        let cflag = subcp orep(a,i,carry)
        and vflag = scovfl orep(a,i,carry)  in
         let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
         and new_psw = (psw except [vf,vflag;cf,cflag])orep
         and new_pc = inc orep pc  in
          Macro_State new_reg new_psw new_pc mem pa2)
```

The SUBCI instruction performs the same actions as the SUBI instruction execpt the carry bit is included in the subtraction.

```
M_BANDI =
⊢def ∀orep mrep Rd Ra imm s e.
    M_BANDI orep mrep Rd Ra imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s   in
      let a = FETCH_REG orep reg Ra pa2
      and i = extend orep imm  in
       let result = band orep(a,i)   in
        let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
        and new_pc = inc orep pc  in
         Macro_State new_reg psw new_pc mem pa2)
```

The BANDI instruction stores into the `Rd` register the bitwise conjunction of the contents of the `Ra` register and the immediate.

```
M_BORI =
⊢def ∀orep mrep Rd Ra imm s e.
    M_BORI orep mrep Rd Ra imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let a = FETCH_REG orep reg Ra pa2
      and i = extend orep imm   in
       let result = bor orep(a,i) in
        let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
        and new_pc = inc orep pc in
         Macro_State new_reg psw new_pc mem pa2)
```

The BORI instruction stores into `Rd` the bitwise disjunction of the contents of the `Ra` register and the immediate.

```
M_BXORI =
⊢def ∀orep mrep Rd Ra imm s e.
    M_BXORI orep mrep Rd Ra imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let a = FETCH_REG orep reg Ra pa2
      and i = extend orep imm  in
       let result = bxor orep(a,i)  in
        let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
        and new_pc = inc orep pc in
         Macro_State new_reg psw new_pc mem pa2)
```

The BXORI instruction stores into `Rd` the bitwise exclusive disjunction of the contents of the `Ra` register and the immediate.

```
M_ADD =
⊢def ∀orep mrep Rd Ra Rb s e.
    M_ADD orep mrep Rd Ra Rb s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let a = FETCH_REG orep reg Ra pa2  in
       let b = FETCH_REG orep reg Rb pa2 in
        let result = add orep(a,b) in
         let cflag = addp orep(a,b)
         and vflag = aovfl orep(a,b) in
          let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
          and new_psw = (psw except [vf,vflag;cf,cflag])orep
          and new_pc = inc orep pc in
           Macro_State new_reg new_psw new_pc mem pa2)
```

The ADD instruction stores into the `Rd` register the sum of the contents of the `Ra` and `Rb` registers.

```
M_ADDC =
⊢def ∀orep mrep Rd Ra Rb s e.
    M_ADDC orep mrep Rd Ra Rb s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let a = FETCH_REG orep reg Ra pa2  in
       let b = FETCH_REG orep reg Rb pa2  in
        let carry = get_cf orep psw in
         let result = addc orep(a,b,carry) in
          let cflag = addcp orep(a,b,carry)
          and vflag = acovfl orep(a,b,carry)  in
           let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
           and new_psw = (psw except [vf,vflag;cf,cflag])orep
           and new_pc = inc orep pc in
            Macro_State new_reg new_psw new_pc mem pa2)
```

The ADDC instruction stores into the Rd register the sum of the contents of the Ra and Rb registers. The add is performed using the carry bit.

```
M_SUB =
⊢def ∀orep mrep Rd Ra Rb s e.
    M_SUB orep mrep Rd Ra Rb s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let a = FETCH_REG orep reg Ra pa2   in
      let b = FETCH_REG orep reg Rb pa2  in
      let result = sub orep(a,b)  in
      let cflag = subp orep(a,b)
         and vflag = sovfl orep(a,b) in
          let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
          and new_psw = (psw except [vf,vflag;cf,cflag])orep
          and new_pc = inc orep pc  in
           Macro_State new_reg new_psw new_pc mem pa2)
```

The SUB instruction stores into the Rd register the difference of the contents of the Ra and Rb registers.

```
M_SUBC =
⊢def ∀orep mrep Rd Ra Rb s e.
    M_SUBC orep mrep Rd Ra Rb s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let a = FETCH_REG orep reg Ra pa2  in
      let b = FETCH_REG orep reg Rb pa2  in
      let carry = get_cf orep psw  in
      let result = subc orep(a,b,carry)  in
      let cflag = subcp orep(a,b,carry)
          and vflag = scovfl orep(a,b,carry)  in
           let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
           and new_psw = (psw except [vf,vflag;cf,cflag])orep
           and new_pc = inc orep pc  in
            Macro_State new_reg new_psw new_pc mem pa2)
```

The SUBC instruction stores into the Rd register the difference of the contents of the Ra and Rb registers. The subtraction is performed including the carry bit.

```
M_BAND =
⊢def ∀orep mrep Rd Ra Rb s e.
    M_BAND orep mrep Rd Ra Rb s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let a = FETCH_REG orep reg Ra pa2  in
       let b = FETCH_REG orep reg Rb pa2  in
        let result = band orep(a,b)  in
         let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
         and new_pc = inc orep pc in
          Macro_State new_reg psw new_pc mem pa2)
```

The BAND instruction stores into `Rd` the bitwise conjunction of the contents of the `Ra` and `Rb` registers.

```
M_BOR =
⊢def ∀orep mrep Rd Ra Rb s e.
    M_BOR orep mrep Rd Ra Rb s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let a = FETCH_REG orep reg Ra pa2  in
       let b = FETCH_REG orep reg Rb pa2 in
        let result = bor orep(a,b)  in
         let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
         and new_pc = inc orep pc  in
          Macro_State new_reg psw new_pc mem pa2)
```

The BOR instruction stores into the `Rd` register the bitwise disjunction of the contents of `Ra` and `Rb`.

```
M_BXOR =
⊢def ∀orep mrep Rd Ra Rb s e.
    M_BXOR orep mrep Rd Ra Rb s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s
     in
      let a = FETCH_REG orep reg Ra pa2
      in
       let b = FETCH_REG orep reg Rb pa2
       in
        let result = bxor orep(a,b)
        in
         let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
         and new_pc = inc orep pc
         in
          Macro_State new_reg psw new_pc mem pa2)
```

The BXOR instruction stores into the `Rd` register the bitwise exclusive disjunction of the contents of `Ra` and the immediate.

```
M_BNOT =
⊢def ∀orep mrep Rd Ra Rb s e.
    M_BNOT orep mrep Rd Ra s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let a = FETCH_REG orep reg Ra pa2  in
       let result = bnot orep a  in
        let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
        and new_pc = inc orep pc  in
         Macro_State new_reg psw new_pc mem pa2)
```

The BNOT stores the bitwise complement of the contents of `Ra` into the register `Rd`.

```
M_LSL =
⊢def ∀orep mrep Rd Ra s e.
    M_LSL orep mrep Rd Ra s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let a = FETCH_REG orep reg Ra pa2 in
       let result = shl orep(a,1)  in
        let cflag = msb orep a in
         let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
         and new_psw = (psw except [cf,cflag])orep
         and new_pc = inc orep pc in
          Macro_State new_reg new_psw new_pc mem pa2)
```

The LSL stores in the register `Rd` the value of the contents of the register `Ra` shifted left. The carry flag in the program status word is set to the value of the bit shifted out.

```
M_LSR =
⊢def ∀orep mrep Rd Ra s e.
    M_LSR orep mrep Rd Ra s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let a = FETCH_REG orep reg Ra pa2  in
       let result = shr orep(a,1) in
        let cflag = lsb orep a   in
         let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
         and new_psw = (psw except [cf,cflag])orep
         and new_pc = inc orep pc  in
          Macro_State new_reg new_psw new_pc mem pa2)
```

The LSR instruction stores into the `Rd` register the value of the contents of the `Ra` register shifted right.

```
M_ASR =
⊢def ∀orep mrep Rd Ra s e.
    M_ASR orep mrep Rd Ra s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let a = FETCH_REG orep reg Ra pa2  in
       let result = setbit orep(msbv,shr orep(a,1),bit orep msbv a)   in
        let cflag = lsb orep a in
         let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
         and new_psw = (psw except [cf,cflag])orep
         and new_pc = inc orep pc  in
          Macro_State new_reg new_psw new_pc mem pa2)
```

The ASR instruction stores into the `Rd` register the value of the contents of the `Ra` register arithmetically shifted right.

```
M_BSL =
⊢def ∀orep mrep Rd Ra s e.
    M_BSL orep mrep Rd Ra s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let a = FETCH_REG orep reg Ra pa2   in
       let result = shl orep(a,8)   in
        let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
        and new_pc = inc orep pc  in
         Macro_State new_reg psw new_pc mem pa2)
```

The BSL instruction stores into the `Rd` register the value of the contents of the `Ra` register byte shifted left.

```
M_BSR =
⊢def ∀orep mrep Rd Ra s e.
    M_BSR orep mrep Rd Ra s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s in
      let a = FETCH_REG orep reg Ra pa2  in
       let result = shr orep(a,8)  in
        let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
        and new_pc = inc orep pc  in
         Macro_State new_reg psw new_pc mem pa2)
```

The BSR instruction stores into `Rd` the value of the contents of the `Ra` register byte shifted right.

```
M_ABSR =
⊢def ∀orep mrep Rd Ra s e.
    M_ABSR orep mrep Rd Ra s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let a = FETCH_REG orep reg Ra pa2  in
       let result =
             setrange
             orep
             (shr orep(a,8))
             (31,24)
             (repeat orep(8,bit orep msbv a))  in
        let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
        and new_pc = inc orep pc  in
         Macro_State new_reg psw new_pc mem pa2)
```

The ABSR instruction stores into the `Rd` register the value of the contents of the `Ra` register arithmetically shifted right.

```
M_SPA =
⊢def ∀orep mrep imm s e.
    M_SPA orep mrep imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let new_pa = bit orep 0(extend orep imm)  in
       let new_psw =
             (psw except [pa,bit orep lsbv(extend orep imm)])orep  in
        let new_pc = inc orep pc in
         Macro_State reg new_psw new_pc mem new_pa)
```

The SPA instruction sets the page address flip-flop to the value of the lowest bit of the immediate.

```
M_RPA =
⊢def ∀orep mrep Rd s e.
    M_RPA orep mrep Rd s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s
     in
      let result =
            ((repeat orep(32,F)) except [lsbv,bit orep pa psw])orep
      in
       let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw result
       and new_pc = inc orep pc
       in
        Macro_State new_reg psw new_pc mem pa2)
```

The RPA instruction stores the value of the page address bit into the lower order bit of `Rd`. The upper 31 bits of `Rd` are set low.

```
M_GPSW =
⊢def ∀orep mrep Rd s e.
    M_GPSW orep mrep Rd s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw psw
      and new_pc = inc orep pc in
       Macro_State new_reg psw new_pc mem pa2)
```

The GPSW instruction stores the program status word into `Rd`.

```
M_PPSW =
⊢def ∀orep mrep Ra s e.
    M_PPSW orep mrep Ra s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s
     in
      let smv = get_sm orep psw
      in
       let ra = FETCH_REG orep reg Ra pa2
       in
        let new_psw = (smv →  ra | psw)
        and new_pc = inc orep pc
        in
         Macro_State reg new_psw new_pc mem pa2)
```

The PPSW instruction stores the contents of register `Ra` into the program status word. The processor must be in supervisory mode for this operation to take effect.

```
M_INT =
⊢def ∀orep mrep Rd Ra imm s e.
    M_INT orep mrep Rd Ra imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s
     in
      let a = FETCH_REG orep reg Ra pa2  in
       let i = extend orep imm  in
        let result = band orep(add orep(a,i),wordn orep 255)  in
         let new_psw =
               ((shl orep(psw,8)) except
                [pa,bit orep pa psw;sm,T;ie,F;cf,get_cf orep psw;
                 vf,get_vf orep psw])
               orep  in
          let new_reg =
                MACRO_UPDATE_REG orep reg Rd pa2 psw(inc orep pc)
          and new_pc = result  in
           Macro_State new_reg new_psw new_pc mem pa2)
```

The INT instruction calls an interrupt service routine. The register `Rd` is loaded with the value of the program counter marking the return point.

```
M_RTI =
⊢def ∀orep mrep Ra imm s e.
    M_RTI orep mrep Ra imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let i = extend orep imm  in
       let a = FETCH_REG orep reg Ra pa2  in
        let result = add orep(a,i) in
         let new_psw = setbit orep(26,shr orep(psw,8),T)  in
          let new_pc = inc orep pc  in
           Macro_State reg new_psw new_pc mem pa2)
```

The RTI instruction returns from an interrupt by jumping execution to the location `Ra` plus Immediate.

```
M_NOOP =
⊢def ∀orep mrep s e.
    M_NOOP orep mrep s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let new_pc = inc orep pc in Macro_State reg psw new_pc mem pa2)
```

The NOOP instruciton performs no actions except to update the program counter.

```
M_JMP =
⊢def ∀orep mrep imm26 s e.
    M_JMP orep mrep imm26 s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s in
      let new_pc = extend26 orep imm26  in
       Macro_State reg psw new_pc mem pa2)
```

The JMP instruction jumps to the memory location specified by the 26-bit immediate.

```
M_JALI =
M_JALI =
⊢def ∀orep mrep Rd Ra imm s e.
    M_JALI orep mrep Rd Ra imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let a = FETCH_REG orep reg Ra pa2 in
       let i = signextend orep imm in
        let result = add orep(a,i) in
         let new_reg = MACRO_UPDATE_REG orep reg Rd pa2 psw(inc orep pc)
           and new_pc = result  in
          Macro_State new_reg psw new_pc mem pa2)
```

The JALI instruction is an unconditional jump to the memory location `Ra` + Immediate.

```
M_BEQ =
⊢def ∀orep mrep Rd Ra imm s e.
    M_BEQ orep mrep Rd Ra imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let rs1 = FETCH_REG orep reg Ra pa2  in
       let rs2 = FETCH_REG orep reg Rd pa2  in
        let a = inc orep pc  in
         let i = signextend orep imm  in
          let result = add orep(a,i) in
           let new_pc = (eqp orep(rs1,rs2) →  result | inc orep pc)  in
            Macro_State reg psw new_pc mem pa2)
```

The BEQ instruction branches if the value in `Rd` is equal to the value in `Ra`. The branch target address is calculated by adding the 16-bit immediate value to the program counter.

```
M_BNEQ =
⊢def ∀orep mrep Rd Ra imm s e.
    M_BNEQ orep mrep Rd Ra imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s   in
      let rs1 = FETCH_REG orep reg Ra pa2  in
       let rs2 = FETCH_REG orep reg Rd pa2 in
        let a = inc orep pc  in
         let i = signextend orep imm  in
          let new_address = add orep(a,i)  in
           let new_pc = ((¬eqp orep(rs1,rs2)) →  new_address | a) in
            Macro_State reg psw new_pc mem pa2)
```

The BNEQ executes exactly like the BEQ instruction except that it branches if the values in `Rd` and `Ra` are not equal.

```
M_BGTZ =
⊢def ∀orep mrep Rd Ra imm s e.
    M_BGTZ orep mrep Rd imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let rd = FETCH_REG orep reg Rd pa2   in
       let i = signextend orep imm   in
        let new_address = add orep(inc orep pc,i)  in
         let new_pc = (gtzp orep rd →  new_address | inc orep pc)  in
          Macro_State reg psw new_pc mem pa2)
```

The BGTZ instruction branches if the value in `Rd` is greater than zero.

```
M_BGTEZ =
⊢def ∀orep mrep Rd Ra imm s e.
    M_BGTEZ orep mrep Rd imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let rd = FETCH_REG orep reg Rd pa2  in
       let i = signextend orep imm  in
        let new_address = add orep(inc orep pc,i)  in
         let new_pc = (gtezp orep rd →  new_address | inc orep pc)   in
          Macro_State reg psw new_pc mem pa2)
```

The BGTEZ instruction branches if the value in the `Rd` register is greater than or equal to zero.

```
M_BLTZ =
⊢def ∀orep mrep Rd Ra imm s e.
    M_BLTZ orep mrep Rd imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let rd = FETCH_REG orep reg Rd pa2  in
       let i = signextend orep imm in
        let new_address = add orep(inc orep pc,i)  in
         let new_pc = (ltzp orep rd →  new_address | inc orep pc)   in
          Macro_State reg psw new_pc mem pa2)
```

The BLTZ instruction branches if the value in the `Rd` register is less than zero.

```
M_BLTEZ =
⊢def ∀orep mrep Rd Ra imm s e.
    M_BLTEZ orep mrep Rd imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let rd = FETCH_REG orep reg Rd pa2  in
       let i = signextend orep imm  in
        let new_address = add orep(inc orep pc,i)  in
         let new_pc = inc orep pc  in
          let new_pc' = (ltezp orep rd →  new_address | inc orep pc)   in
           Macro_State reg psw new_pc' mem pa2)
```

The BLTZ instruction branches if the value in the `Rd` register is less than or equal to zero.

```
M_BC =
⊢def ∀orep mrep s e imm.
    M_BC orep mrep imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s   in
      let i = signextend orep imm   in
       let new_address = add orep(inc orep pc,i)  in
        let new_pc = (get_cf orep psw →  new_address | inc orep pc)  in
         Macro_State reg psw new_pc mem pa2)
```

The BC instruction branches if the carry bit of the program status word is high. Once again the branch target is calculated by adding the value of the immediate to the program counter.

```
M_BOV =
⊢def ∀orep mrep Rd Ra imm s e.
    M_BOV orep mrep imm s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s  in
      let i = signextend orep imm   in
       let new_address = add orep(inc orep pc,i)  in
        let new_pc = (get_vf orep psw →  new_address | inc orep pc)  in
         Macro_State reg psw new_pc mem pa2)
```

The BOV instruction branches if the overflow flag in the program status word is set high.

```
⊢def ∀orep mrep s e.
    M_EINT orep mrep s e =
    (let reg = Reg s
     and psw = Psw s
     and pc = Pc s
     and mem = Mem s
     and pa2 = Pa s
     in
      let a = FETCH_REG orep reg R0 pa2 in
      let i = extend orep(low16w orep(Ivecp e)) in
      let result = band orep(add orep(a,i),wordn orep 255) in
      let new_psw =
               ((shl orep(psw,8)) except
                [pa,get_pa orep psw;sm,T;ie,F;cf,get_cf orep psw;
                 vf,get_vf orep psw])
               orep  in
      let new_reg =
                MACRO_UPDATE_REG orep reg R1 pa2 psw(inc orep pc) in
      let new_pc = result in
      let new_pa = get_pa orep psw in
             Macro_State new_reg new_psw new_pc mem new_pa)
```

Appendix E

The ALU VHDL

In this appendix we present the VHDL code for the ALU which was left out of the main text.

```
do_alu : PROCESS(mx4f,mx4g,op2,psw,carry)

  VARIABLE temp : qsim_state_vector(32 DOWNTO 0);
  --VARIABLE tempint : integer range 0 to 1;
  VARIABLE tempint0 : qsim_state_vector(32 DOWNTO 0);
  VARIABLE tempint1 : qsim_state_vector(32 DOWNTO 0);
  VARIABLE tempint2 : qsim_state_vector(32 DOWNTO 0);

  VARIABLE templ: qsim_state_vector(32 DOWNTO 0);
  VARIABLE aa   : qsim_state_vector(31 DOWNTO 0);
  VARIABLE bb   : qsim_state_vector(31 DOWNTO 0);
  BEGIN
  aa := mx4f;
  IF ((op2(5 DOWNTO 3) = "001" OR op2(5 DOWNTO 3) = "010") AND
<      (op2(2 DOWNTO 1) = "01")) THEN
     bb := NOT mx4g;
  ELSE
     bb := mx4g;
  END IF;
  -- Set some good defaults
  npsw <= psw;
  --tempint := to_integer('0' & carry,0);
  --temp(16 DOWNTO 0) := to_qsim_state(to_integer('0'&aa(15 DOWNTO 0),0)
  --                    + to_integer('0'&bb(15 DOWNTO 0),0
  --                    + tempint),17);
  --tempint := to_integer('0'&temp(16),0);
  --temp(32 DOWNTO 16) := to_qsim_state(to_integer('0'&aa(31 DOWNTO 16),0)
  --                    + to_integer('0'&bb(31 DOWNTO 16),0)
  --                    + tempint,17);
  tempint0 := "00000000000000000000000000000000"&carry;
  tempint1:= '0'&aa;
  -- tempint1(32) := '0';
  tempint2 := '0'&bb;
```

```
-- tempint2(32) := '0';
temp := tempint2 + tempint1 + tempint0;
alu <= temp(31 DOWNTO 0);

IF (op2(5 DOWNTO 3) = "001" OR op2(5 DOWNTO 3) = "010") THEN -- ALU and ALUI
  CASE op2(2 DOWNTO 0) IS
    WHEN "000" => -- ADD and ADDI
      alu <= temp(31 DOWNTO 0);
      npsw(1) <= temp(32); --carry
      npsw(0) <= (aa(31) AND bb(31) AND NOT temp(31)) OR
                 (NOT aa(31) AND NOT bb(31) AND temp(31)); -- Overflow
    WHEN "001" => -- ADDC and ADDCI
      alu <= temp(31 DOWNTO 0);
      npsw(1) <= temp(32); --carry
      npsw(0) <= (aa(31) AND bb(31) AND NOT temp(31)) OR
                 (NOT aa(31) AND NOT bb(31) AND temp(31)); -- Overflow
    WHEN "010" => -- SUB and SUBI
      alu <= temp(31 DOWNTO 0);
      npsw(1) <= temp(32); --carry
      npsw(0) <= (aa(31) AND NOT bb(31) AND NOT temp(31)) OR
                 (NOT aa(31) AND bb(31) AND temp(31)); -- Overflow
    WHEN "011" => -- SUBC and SUBCI
      alu <= temp(31 DOWNTO 0);
      npsw(1) <= temp(32); --carry
      npsw(0) <= (aa(31) AND NOT bb(31) AND NOT temp(31)) OR
                 (NOT aa(31) AND bb(31) AND temp(31)); -- Overflow
    WHEN "100" =>
      alu <= (aa AND bb);
    WHEN "101" =>
      alu <= (aa OR bb);
    WHEN "110" =>
      alu <= (aa XOR bb);
    WHEN "111" =>
      alu <= NOT aa;
    WHEN OTHERS => -- Must have received X's or Z's as inputs
      alu <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
  END CASE;
ELSIF (op2(5 DOWNTO 3) = "011") THEN -- shift instr.
  CASE op2(2 DOWNTO 0) IS
    WHEN "000" => -- LSL
      templ(31 DOWNTO 0) := aa;
      alu(31 DOWNTO 1) <= templ(30 DOWNTO 0);
      alu(0) <= '0';
      npsw(1) <= aa(31);
    WHEN "001" => -- LSR
      templ(31 DOWNTO 0) := aa;
      alu(30 DOWNTO 0) <= templ(31 DOWNTO 1);
      alu(31) <= '0';
      npsw(1) <= aa(0);
    WHEN "010" => -- ASR
      templ(31 DOWNTO 0) := aa;
      alu(30 DOWNTO 0) <= templ(31 DOWNTO 1);
      alu(31) <= templ(31);
      npsw(1) <= aa(0);
```

```
    WHEN "011" => -- BSL
      templ(31 DOWNTO 0) := aa;
      alu(31 DOWNTO 8) <= templ(23 DOWNTO 0);
      alu(7 DOWNTO 0) <= "00000000";
    WHEN "100" => -- BSR
      templ(31 DOWNTO 0) := aa;
      alu(23 DOWNTO 0) <= templ(31 DOWNTO 8);
      alu(31 DOWNTO 24) <= "00000000";
    WHEN "101" => -- ABSR
      templ(31 DOWNTO 0) := aa;
      alu(23 DOWNTO 0) <= templ(31 DOWNTO 8);
      alu(31 DOWNTO 24) <= templ(31) & templ(31) & templ(31) & templ(31) &
                           templ(31) & templ(31) & templ(31) & templ(31);
    WHEN "110" =>
      alu <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
      NULL; -- Not a valid instr.
    WHEN "111" =>
      alu <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
      NULL; -- Not a valid instr.
    WHEN OTHERS => -- Must be receiving X's or Z's
      alu <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
  END CASE;
ELSIF (op2(5 DOWNTO 3) = "000") THEN -- special instr.
  CASE op2(2 DOWNTO 0) IS
    WHEN "000" =>
      NULL; -- nop
    WHEN "001" => -- SPA
      npsw(4) <= bb(0);
    WHEN "010" => -- RPA
      alu(31 DOWNTO 1) <= "0000000000000000000000000000000";
      alu(0) <= psw(4);
    WHEN "011" => -- GPSW
      alu <= psw;
    WHEN "100" => -- PPSW
      IF (psw(3) = '1') THEN
        npsw <= aa;
      END IF;
    WHEN "101" => -- INT
      alu(7 DOWNTO 0) <= temp(7 DOWNTO 0);
      alu(31 DOWNTO 8) <= "000000000000000000000000";
      npsw(31 DOWNTO 8) <= psw(23 DOWNTO 0);
      npsw(7 DOWNTO 4) <= psw(7 DOWNTO 4);
      npsw(3 DOWNTO 2) <= "10";
      npsw(1 DOWNTO 0) <= psw(1 DOWNTO 0);
    WHEN "110" => -- RTI
      alu <= temp(31 DOWNTO 0);
      npsw(31 DOWNTO 24) <= "00000100";
      npsw(23 DOWNTO 0) <= psw(31 DOWNTO 8);
    WHEN "111" => -- INVALID
      alu <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
    WHEN OTHERS =>
      alu <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
  END CASE;
```

```
ELSIF (op2(5 DOWNTO 3) = "110" OR op2(5 DOWNTO 3) = "111") THEN
      -- BRANCH or JUMP type instructions
  alu <= temp(31 DOWNTO 0);
ELSIF (op2(5 DOWNTO 3) = "100") THEN -- LD, ST, LHI
  CASE op2(2 DOWNTO 0) IS
    WHEN "000" => -- LD
      alu <= temp(31 DOWNTO 0);
    WHEN "001" => -- LHI
      alu(15 DOWNTO 0) <= aa(15 DOWNTO 0);
      alu(31 DOWNTO 16) <= bb(15 DOWNTO 0);
    WHEN "010" => -- ST
      alu <= temp(31 DOWNTO 0);
    WHEN "011" => -- INVALID
      alu <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
    WHEN "100" => -- INVALID
      alu <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
    WHEN "101" => -- INVALID
      alu <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
    WHEN "110" => -- INVALID
      alu <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
    WHEN "111" => -- INVALID
      alu <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
    WHEN OTHERS =>
      alu <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
  END CASE;

ELSE -- Either inputting X's or an invalid instruction
  alu <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
END IF;
END PROCESS do_alu;
```