Verifying Pipelined Microprocessors

Phillip J. Windley

Laboratory for Applied Logic
Department of Computer Science
Brigham Young University
Provo, Utah 84602-6576

windley@cs.byu.edu

Abstract

Recently there has been much research in verifying pipelined microprocessors. Even so, there has been little consensus on what form the correctness statement should take. Put another way, what should we be verifying about pipelined microprocessors? We believe that the correctness statement should show that the parallel machine represented by the pipeline behaves in the same manner as the sequential machine represented by the instruction set semantics. In this paper, we present such a model and examine four pipeline verifications to see how they compare.

1 Introduction

Much has been written over the years regarding the formal specification and verification of microprocessors. Most of these efforts have been directed at non–pipelined microprocessors. In nearly all of these proofs, the final results, or correctness statement had the form:

$$\forall s.\, \mathcal{I}(s) \;\Rightarrow\; \mathcal{B}(\mathcal{A}(s))$$

where s is the state stream, $\mathcal{I}$ is a predicate modeling the implementation, $\mathcal{B}$ is a predicate representing the desired behavior, and $\mathcal{A}$ is an abstraction function relating the implementational state stream to behavioral state stream.

Over the last several years, several pipelined microprocessors have been subjected to formal verification. There are as many different correctness statements (i.e. final results) as verifications.

What does it mean for a pipelined microprocessor to be *correct*? What was, in fact, proven? Do the notions of correctness used in these efforts correspond to each other in any way?

This paper discusses a correctness statement for pipelined microprocessors and discusses four recent verification of pipelined microprocessors including an example verification using our correctness statement. We examine how the correctness statements used for the example pipelines differ from the one proposed in this paper.

This paper will closely examine the verification results of four pipelined microprocessors. We do not focus on the technique (which follows, to some extent, the form of the correctness statement), but rather concentrate on the correctness statement itself. The four pipelines that we examine are the Saxe pipeline verification by Saxe, et al. [SHGG93], the DLX verification by Tahar and Kumar [TK93], the DLX verification by Burch and Dill [BD94], and our own Uinta verification [WC94].

These four proofs are done using three different theorem provers: the Tahar DLX proof and the Uinta proof were both done in HOL [GM93], the Burch DLX proof was done using a model checker, and the Saxe pipeline was verified in the Larch Prover [GG91]. In order to more fully understand the correctness statements and methodology used in each proof, we first translated them into HOL and then reverified them using the original author's correctness statement. This led to considerable insight into the correctness statements themselves and to their effect on the proof methodology used to prove them.

This paper will proceed as follows: after a brief introduction to the HOL logic, we will discuss correctness statements in non–pipelined and pipelined machines and present the correctness statements for each of the pipelines. After the correctness statements have been presented, we will compare them and discuss strengths and weaknesses [1].

2 A Brief Introduction to HOL.

We have redone each of the pipeline verifications presented in this paper using the HOL theorem proving system in order to provide a common notation and framework for discussion. 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 making it easy to express properties from the other systems in HOL. Furthermore HOL is widely available, robust, and has a growing world-wide user base. However, there is nothing in our work that requires the HOL theorem proving system.

[1]We should point out that the main goal of most of the work in which these correctness statements appear was *not* defining pipeline correctness, but rather the exposition of a proof technique for pipelines. We are not, therefore, calling into question the work itself, but rather the choice of the correctness statement for the pipeline proof.

HOL is a general theorem proving system developed at the University of Cambridge [GM93] 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`." A statement that would read "if `a`, then `b`, else if `c` then `d` else if ... else `e`" would appear in HOL as

```
a → b |
c → d |
...   | e
```

- The construct `let v1 = expr1 and v2 = expr2 and ... in` defines local variables `v1`, `v2`, etc. with values `expr1`, `expr2`, etc. simultaneously.

3 Microprocessor Correctness

The essence of verification is to relate mathematical models at different levels of abstraction. In the discussion that follows, and for the rest of the paper, we speak of the "abstract level" and the "concrete level," but these terms are relative; as we move up and down a hierarchy of interpreters, what we call "abstract" at one level will be termed "concrete" with respect to the level above it. As a matter of convention, we will decorate variables that represent the concrete level with primes.

3.1 Instruction Set Semantics

The instruction set semantics is given by describing the state change affected by the instructions. The specification of an instruction set semantics is largely the same as any language semantics specification. Some are given in an operational manner and others using a denotational approach. This section will describe in very general terms how the instruction set semantics of a microprocessor can be specified. For a more detailed account see [Win94].

3.1.1 State

An important consideration in defining the instruction set semantics is to carefully define the semantic domain, or *state*, of the microprocessor. For now we will treat state as an uninterpreted object of type $\mathbf{S}$ (i.e. nothing is known of its structure or content). We write $\mathbf{S} \leq \mathbf{S}'$ to indicate that $\mathbf{S}$ is an abstraction of $\mathbf{S}'$.

We are more interested in sequences of states than individual states. We will represent a state stream, $\mathbf{U}$, as a function from time to state, $\mathbf{N} \rightarrow \mathbf{S}$.

3.2 Environments

The environment represents the external world; it is where interrupt requests originate, reset signals are generated, and so on. At the abstract level, we treat the environment as an uninterpreted type. We denote it as $\mathbf{E}$.

3.2.1 The Interpreter Specification

For purposes of this section, we describe the semantics of the instruction set using an interpreter. Interpreters are state transition systems. An interpreter, $\mathcal{I}$, is a predicate defined in terms of a 3–tuple, $\mathcal{J}, \mathcal{K}, \mathcal{C}$, where $\mathcal{J}$, $\mathcal{K}$, and $\mathcal{C}$ are defined as follows:

- Let $\mathbf{J}$ be the set of all functions with domain $(\mathbf{S} \times \mathbf{E})$ and codomain $\mathbf{S}$. Not all functions in $\mathbf{J}$ are meaningful; the specifier's job is to choose meaningful functions. We use a subset of $\mathbf{J}$ to represent the instruction set; we call this set $\mathcal{J}$. The functions in $\mathcal{J}$ provide a denotational semantics for the instructions that they represent.
- In order to uniquely identify each instruction in $\mathcal{J}$, we associate it with a unique key. At the abstract level, we take keys from the uninterpreted domain, $\mathbf{K}$. We must be able to choose instructions from $\mathcal{J}$ according to some predefined selection criteria. The selection is based on the current state and environment. We define $\mathcal{K}$ to be a function with domain $(\mathbf{S} \times \mathbf{E})$ and codomain $\mathbf{K}$.
- We define $\mathcal{C}_{\mathcal{J}}$ to be a choice function that has domain $\mathbf{K}$ and codomain $(\mathbf{S} \times \mathbf{E} \rightarrow \mathbf{S})$. $\mathcal{C}_{\mathcal{J}}$ selects the state transition function from $\mathbf{J}$ that is associated with key $\mathbf{K}$.

We define an interpreter, $\mathcal{I}[s, e]$, as a predicate over the state stream, s, and the environment stream, e. The definition of $\mathcal{I}$ is given as

$$\mathcal{I}[s, e] \;\equiv\; \forall t : \mathbf{N}.\ s_{t+1} = \mathcal{N}(s_t,\ e_t)$$

where

$$\mathcal{N} = \mathcal{C}_{\mathcal{J}}(k_t)$$

and

$$k_t = \mathcal{K}(s_t, e_t)$$

In this equation, s_t (e_t) is the state (environment) in the state (environment) stream s (e) at time t. The predicate, $\mathcal{I}$, constrains the state of the interpreter at time $t+1$ to be a function, $\mathcal{N}$, of the state and environment at time t. The function is determined by applying the choice function, $\mathcal{C}_{\mathcal{J}}$, to the key returned by $\mathcal{K}$ for the state and environment at time t.

3.3 The Correctness Statement

Verifying a microprocessor entails showing that the instruction set semantics follows from some lower level description of the microprocessor. This lower level description is typically a less abstract behavioral level although structural descriptions of the circuit can be used.[2] For all but the lowest level description (which is typically structural), the concrete level in the verification can be treated as an interpreter as well.

Showing the behavior specified by the abstract interpreter follows from the behavior specified by the concrete interpreter depends on being able to define the state and environment streams for the abstract interpreter in terms of the state and environment streams for the concrete interpreter. In general terms, we define $\mathcal{A}$ such that

$$s = \mathcal{A}(s')$$

where s is the state stream at the abstract level and s' is the state stream at the concrete level. The abstraction embodied in $\mathcal{A}$ represents the architect's view of how the state in the implementation maps to the state for the instruction set semantics.

There are two kinds of abstraction taking place between different interpreters: data and temporal.

The data abstraction occurs because the state at the abstract level does not typically contain all of the state needed to operate the more detailed interpreter. For example, most microprocessors have an instruction register, but no mention of the instruction register is made at the instruction set level.

[2]We show in [Win90] that for all but the smallest microprocessors, a hierarchical decomposition is needed to decrease the semantic distance between descriptions and make the proof tractable.

The temporal abstraction occurs because the events that the are visible at the concrete level do not always happen in the same order or in the same number of steps that are visible at the abstract level.

Our goal is to prove a correctness relation between the interpreters at different levels of a microprocessor abstraction. In particular, for two interpreters, $\mathcal{I}_m$ and $\mathcal{I}_\ell$, we wish to show that

$$\mathcal{I}_m[s_m, e_m] \;\Rightarrow\; \mathcal{I}_\ell[\mathcal{A}_s(s_m), \mathcal{A}_e(e_m)]$$

where s_m (e_m) is the state (environment) stream at level m and $\mathcal{A}_s$ ($\mathcal{A}_e$) is the state (environment) abstraction function between these two levels. When this implication is true, $\mathcal{I}_\ell$ is an abstraction of $\mathcal{I}_m$ and $\mathcal{I}_m$ is said to *implement* $\mathcal{I}_\ell$.

The form of the correctness statement is very important.[3] We do not believe that microprocessor verification is an end to itself, but rather one step in a chain of verifications that ultimately give confidence that programs operating on a particular implementation behave as expected. Seen in this light, it is important to ensure that the form of the correctness statement is complete and usable in other contexts. For example, in [Win91] we show that verifying security properties of a microprocessor can be easily done from the architectural level and then combined with a proof that the circuit implements the architectural level to show that the circuit has those properties as well.

In another example, we are currently building a verified OS kernel on top of the verified microprocessor Uinta. That work is made easier by the fact that Uinta uses a standard correctness statement like the one shown above.

The correctness statement for both pipelined and non–pipelined microprocessors takes the form shown above. The difference lies in the abstraction function, $\mathcal{A}$. The following sections discuss the differences.

3.4 Abstraction in Non–Pipelined Processors

For a non—pipelined microprocessor, the instruction boundaries in the abstract and concrete interpreters are coincident (see Figure 1). Even so, the concrete level takes many more steps in-between instruction boundaries; thus, there is a temporal abstraction between the two levels.

What makes verification of a non–pipelined microprocessor relatively easy is that the data and temporal abstraction are orthogonal. We can define a data abstraction function that maps members of the concrete state to members of the abstract state and a separate temporal abstraction function that selects members of the state stream that fall at instruction boundaries. The

[3] We are not arguing for predicate–based correctness statements over function–based correctness statements. One could do all of this in a functional context with the same results.

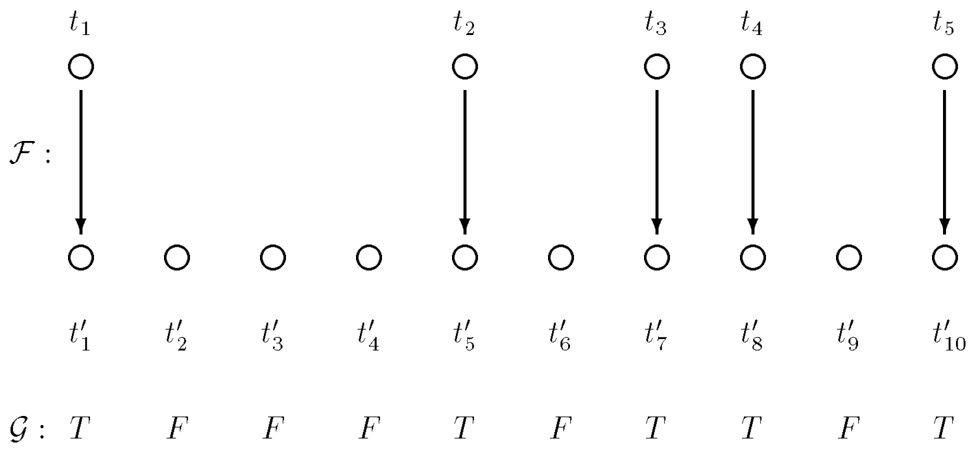

Figure 1: The function, $\mathcal{F}$, which maps time at one level to another, can be defined in terms of a predicate, $\mathcal{G}$, which is true only at instruction boundaries.

state abstraction function, $\mathcal{S}$, typically selects members of the concrete state and reaggregates them to form the abstract state.

Defining the temporal abstraction function can be more involved. In general, different levels in the interpreter hierarchy have different views of time. A temporal abstraction function maps time at the abstract level to time at the concrete level [Mel88]. Figure 1 shows a temporal abstraction function, $\mathcal{F}$. The circles represent clock ticks. Notice that the number of clock ticks required at the concrete level to produce one clock tick at the abstract level may be irregular.

We define the temporal projection, $\mathcal{F}$, recursively in terms of a predicate, $\mathcal{G}$, which is true whenever there is a valid abstraction from the concrete level to the abstract level. In a non–pipelined microprocessor, $\mathcal{G}$ is usually a predicate indicating an instruction boundary.

The function $\mathcal{F}$ is defined recursively so that $\mathcal{F}(\mathcal{G}, 0)$ is the first time that $\mathcal{G}$ is true and $\mathcal{F}(\mathcal{G}, (n+1))$ is the next time after time n when $\mathcal{G}$ is true. The resulting function is monotonically increasing. We use $\mathbf{N}$ to represent time. Thus, we define $\mathcal{F} : (\mathbf{N} \rightarrow \mathbf{B}) \times \mathbf{N} \rightarrow \mathbf{N}$ such that

$$\forall n\ m.\ (n > m) \ \Rightarrow\ (\mathcal{F}(\mathcal{G}, n) > \mathcal{F}(\mathcal{G}, m))$$

We refer the interested reader to the references given above and to [Win95] for the details of the temporal abstraction function.

We can now define the state stream at the abstract level, u, in terms of the state stream at the concrete level, u'. Using the state abstraction function, $\mathcal{S}$, and the temporal abstraction function, $\mathcal{F}$, we define stream abstraction as follows

$$u \equiv \mathcal{S} \circ u' \circ \mathcal{F}$$

where $\circ$ denotes function composition. We perform the same kinds of abstraction on the environment as on states.

3.5 Abstraction in Pipelined Processors

Ultimately what the user of a pipelined microprocessor would like to know is that the parallel machine that pipeline represents behaves in the same manner as the sequential machine that is denoted by the instruction set semantics. Clearly, a goal of the form

$$\mathcal{I}_m[s_m, e_m] \;\Rightarrow\; \mathcal{I}_\ell[\mathcal{A}_s(s_m), \mathcal{A}_e(e_m)]$$

does that. Nothing else gives the same confidence that the machine behaves as it should. Of the machines examined in this paper, only Uinta has been verified using this exact goal. We will discuss the differences between this goal and ones used in each verification in the respective sections.

The key difference between the abstraction function for a pipelined microprocessor and a non–pipelined microprocessor is that for pipelined microprocessors the data and temporal abstractions in a pipeline are *not* orthogonal. Indeed, the correctness model depends on being able to mix the temporal and data abstractions.

The following discussion gives some idea of what we mean when we say a pipeline is correct and shows how the temporal and data abstractions are mixed. Suppose that we execute the following program fragment:

PC	**Instruction**
y	`ADD x1 x2 x3`
y + 1	`LDI x4 x5 imm1`
y + 2	`SUB x6 x4 x7`
y + 3	`JMP z`

We can picture the execution of the pipe as shown in Figure 2. On the left of the figure are the names of the pipeline stages. On the top is the time relative to the start of the execution of this code fragment. The square labeled `ADD` in the line labeled with `EX` indicates that the execute stage of the pipeline performs the state transitions for the `ADD` instruction between time $t + 2$ and $t + 3$. The labels on the right side of the figure indicate the architectural model states that are updated in the corresponding stage.

There are several points of interest regarding Figure 2:

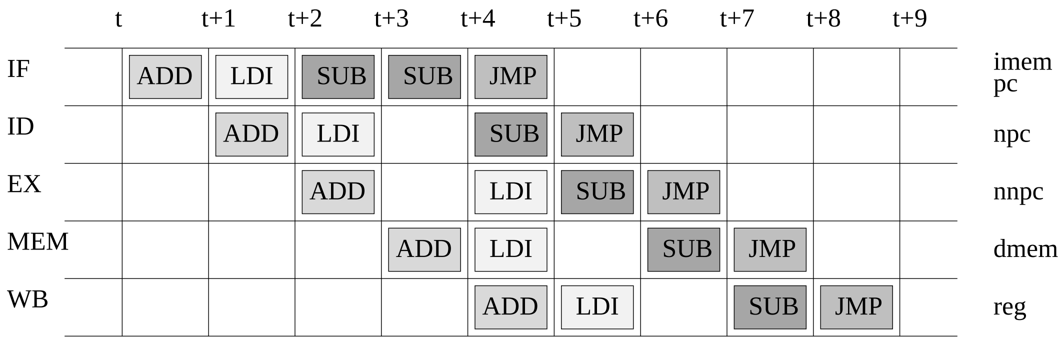

Figure 2: Uinta Pipeline Execution

- The `ADD` instruction updates the data memory between times $t+3$ and $t+4$, for example, and the register file between times $t+4$ and $t+5$. The state transitions indicated by the architectural model are spread out over time. We call this *skew*.
- Instruction execution can be delayed by hazards. The `SUB` instruction must wait to be executed because one of its arguments, `x4`, depends on the value being loaded by the `LDI` instruction. The value will not be ready for forwarding to the ALU in time to avoid a delay. This is called a *stall*. The hardware automatically injects a `NOOP` instruction, represented by blank cells, into the pipeline between the `LDI` and `SUB` instruction.
- The skew is not uniform in the presence of stalls. For example, the effect of the `LDI` instruction on the `nnpc` state variable does not occur between times $t+3$ and $t+4$, as we would expect, but occurs between times $t+4$ and $t+5$. We call this *shifting*.

Because of skewing, stalling, and shifting, the data and temporal abstraction cannot be separated. This makes defining the abstraction function for a pipelined microprocessor much more difficult. Indeed, for a non–pipelined microprocessor, defining the abstraction function can be automated to a large degree [Win93]. We do not know of a similar theory for the less structured abstraction functions needed for verifying pipelines.

4 The Saxe Correctness Statement

In [SHGG93] Saxe et al. describe the verification of a simple pipelined microprocessor. The processor has a five stage pipeline and a simple arithmetic instruction set. There is no notion of an external data memory and consequently no loads and stores. All register file transactions, however, are visible on external ports, `WA_out` and `WD_out`. The processor has branch delay and data forwarding to mitigate data hazards.

Saxe et al. give a top level behavioral description which corresponds to an instruction set semantics, but this description is never used in the verification, so we omit it here. The pipeline

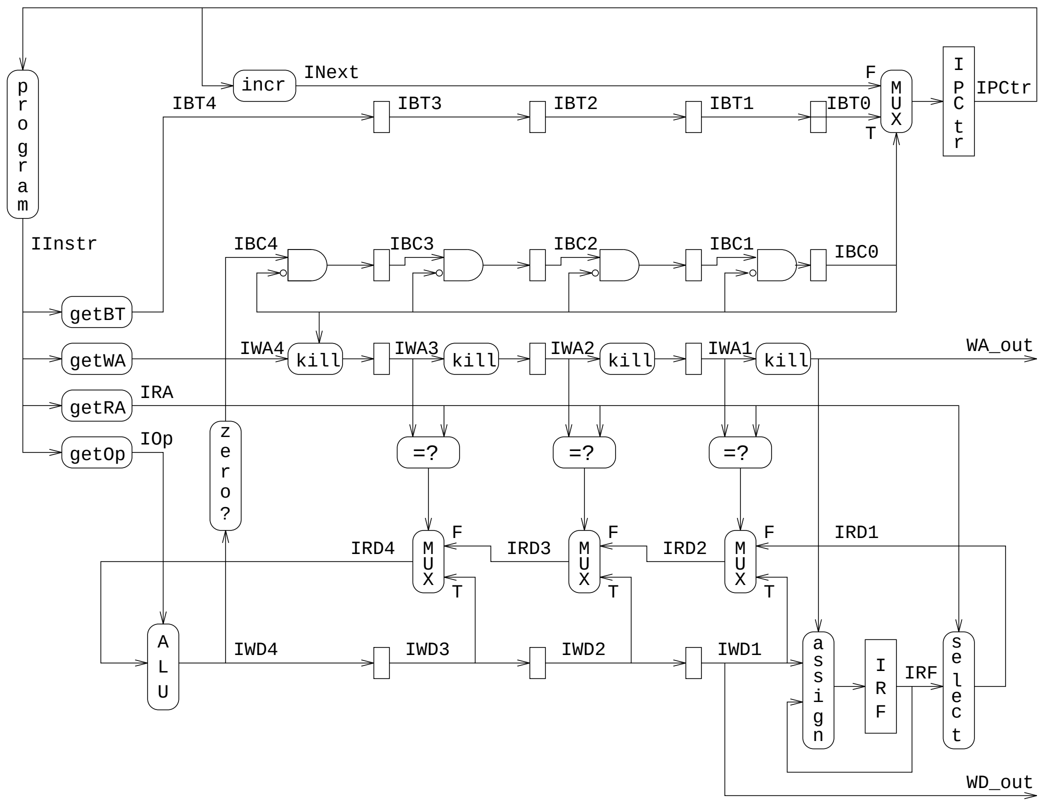

Figure 3: Saxe Pipeline

that is used in the verification is shown in Figure 3. The pipeline is described by a predicate, which we call `Imach`, which defines a state transition function. There is insufficient space in this paper to present the specification of the pipeline.

The abstraction between the top–level state and the pipeline state is described using a predicate rather than a function, but the result is the same as that described in Section 3.5. The top–level state components (identified by the naming convention that their names start with S), are defined in terms of the pipeline level state components (names begin with I). For example, the top–level program counter, `SPCtr`, is defined three time steps in the future since in this pipeline, that is when its effect will be seen.

```
⊢def abs (SInstr, SRA, SWA, SOp, SBT, SRD, SWD, SRF, SBC, SNext, SPCtr)
       (IInstr, IRA, IOp, IRF, IPCtr, IBT3, IBT2, IBT1, IBT0,
        IBC3, IBC2, IBC1, IBC0)
       Stalled =
     ∀ t:time .
     (SRF t = IRF t) ∧
     (SPCtr (t+3) = (IBC0 t) →  (IBT0 t) |
                    (IBC1 t) →  (IBT1 t) |
                    (IBC2 t) →  (IBT2 t) |
                    (IBC3 t) →  (IBT3 t) |
                                (IPCtr t)) ∧
     (SNext t = incr (SPCtr t)) ∧
     (SInstr t = program (SPCtr t)) ∧
     (SBT t = getBT (SInstr t)) ∧
     (SWA t = getWA (SInstr t)) ∧
     (SRA t = getRA (SInstr t)) ∧
     (SOp t = getOp (SInstr t)) ∧
     (SRD t = select (IRF t, SRA t)) ∧
     (SWD t = ALU rep(SOp t, SRD t)) ∧
     (SBC t = zero (SWD t)) ∧
     (SNext t = incr (SPCtr t)) ∧
     (Stalled (t+3) = (IBC0 t) ∨ (IBC1 t) ∨ (IBC2 t) ∨ (IBC3 t))
```

The correctness statement used by Saxe et al. is shown below. It states that the low–level machine, `Imach`, and the abstraction predicate, `abs`, imply four properties. The first defines correct behavior for the register file, the second defines correct behavior for the program counter, the third defines correct behavior for the write address, and the final property defines correct behavior for the write data output.

```
⊢ Imach (IInstr, IRA, IOp, IRF, INext, IPCtr, IRD4, IRD3, IRD2, IRD1,
         IWA4, IWA3, IWA2, IWA1, IWD4, IWD3, IWD2, IWD1, IBT4, IBT3,
         IBT2, IBT1, IBT0, IBC4, IBC3, IBC2, IBC1, IBC0, WA_out, WD_out) ∧
   abs (SInstr, SRA, SWA, SOp, SBT, SRD, SWD, SRF, SBC, SNext, SPCtr)
       (IInstr, IRA, IOp, IRF, IPCtr, IBT3, IBT2, IBT1, IBT0,
        IBC3, IBC2, IBC1, IBC0)
       Stalled ⇒
   (SRF (t+4) = (Stalled (t+3)) →  SRF (t+3)
                               | assign (SRF(t+3), SWA(t+3), SWD(t+3))) ∧
   (SPCtr (t+4) = (Stalled (t+3)) →  SPCtr (t+3)
                                 | (SBC (t+3)) →  (SBT (t+3))
                                 | (SNext (t+3))) ∧
   (WA_out (t+3) = (Stalled (t+3)) →  null | SWA (t+3)) ∧
   ¬(Stalled (t+3)) ⇒  (WD_out (t+3) = (SWD (t+3)))
```

Translating the LP specification of the pipeline to HOL and proving the correctness statement shown above took less than 2 hours. This is not surprising since the LP specification was well laid out and easy to follow. Also, the pipeline is not large and the goal follows easily from the specification.

Discussion. The correctness statement used in this verification, obviously, does not follow the format we lay out in Section 3.3. It is certainly necessary to prove the properties about the register file and program counter that are represented in the correctness statement shown above. We maintain, however, that it is not enough.

The correct result proves properties about only two members of the top–level state. The correctness statement used in the Saxe verification proves interesting properties about the pipeline, but does *not* prove that the pipeline implements the sequential semantics of the top–level specification; indeed, the top–level specification is never used in the proof. At best there is an informal relationship between the top–level specification and the correctness statement.

5 The Tahar Correctness Statement

Tahar and Kumar [TK93] present the verification of a machine similar to the DLX processor of [HP90]. DLX contains a five stage pipeline line, with a two phase clock, and its architecture includes 51 basic instructions. As part of the work done by Tahar and Kumar, DLX was implemented in a commercial VLSI CAD environment using 1.0 μm CMOS technology. The design has approximately 150,000 transistors and occupies a silicon area of 60.34 mm^2.

Because the Tahar verification was already in HOL, we did not redo it. The Tahar specification of the instruction set semantics does not rely on an overall interpreter. Rather, the semantics of each instruction are specified separately by instantiating a specification of the semantics for an entire class of instructions (such as the ALU instructions).

There is not a single correctness statement, but rather two separate theorems which, taken together, form the correctness statement. The first of these theorems looks similar to the correctness statement from Section 3.3:

```
⊢ EBM ⇒  Instruction_Level
```

This theorem proves that the electronic block model (i.e. the structural model) implies the top–level semantics. This proof, however, is done under the assumption that there is only one instruction operating at a time. That is, the first goal does not prove the the parallel machine behaves in the same way as the sequential machine.

The abstraction model used in the proof of the instruction semantics uses orthogonal data and temporal abstractions (see Section 3.4). This works because what is being proven is that the result of the smaller state transitions of each stage of the pipeline result in the same state change as each instruction taken separately.

The second theorem that comprises the correctness statement shows that the electronic block model implies correct instruction pipelining assuming certain software constraints:

$\vdash$ `SW_Constraints` $\wedge$ `EBM` $\Rightarrow$ `Correct_Instr_Pipelining`

Correct instruction pipelining is defined as an absence of conflicts:

$\vdash_{def}$ `Correct_Instr_Pipelining` $=$ $\neg$`Resource_Conflict` $\wedge$ $\neg$`Data_Conflict` $\wedge$ $\neg$`Control_Conflict`

Tahar and Kumar prove absence of conflicts by showing that all possible combinations of n_s instructions do not lead to any conflicts where n_s is the number of stages. The complexity of the proof is managed by splitting the goal into smaller pieces and then combining the results to achieve the end goal. The entire proof script runs in less than 2 hours on a SPARC10.

Discussion. As with the Saxe pipeline verification, the correctness statement in the Tahar and Kumar proof does not follow the format that we outline in Section 3.3. While it is possible that an overall correctness result follows from the two theorems proven by Tahar and Kumar, no such proof exists to our knowledge. Thus, we are left to wonder if proving these two properties separately covers all of the possible errors. Also, as we mentioned earlier, the absence of an overall correctness statement makes using these results in other proofs difficult.

6 The Burch Correctness Statement

Burch and Dill [BD94] report on use of uninterpreted operations in a model checker for verifying pipeline control. There are two pipelines among the examples that they give. The first is a simple three stage pipeline (see Figure 4) that has been used as a benchmark in BDD papers for several years; we will refer to it as the Burch pipeline. The second example is a verification of DLX [HP90]. We have reproduced both of these proofs in HOL; we will use the simple pipeline as the example in this section and comment on the DLX proof at the end of the section.

The Burch pipeline is very simple. The pipeline does not perform instruction fetching or any other memory operations. The instruction inputs (opcode and register indices) are assumed to be available on external lines as is the stall flag. The top–level specification for the Burch pipeline is given in the following HOL definition:

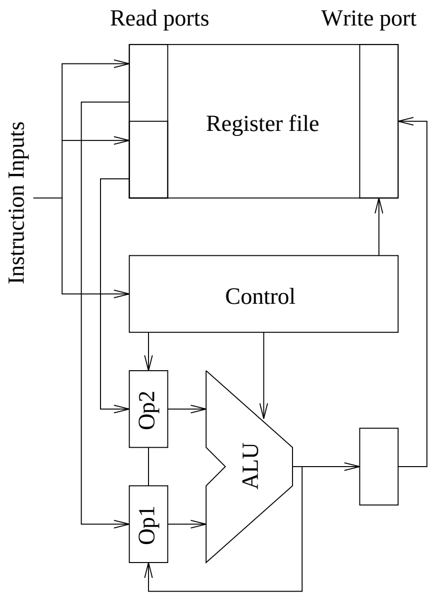

Figure 4: Burch Pipeline

```
⊢def Smach regfile (stall, op, src1, src2, dest) =
          (stall →  regfile
                  | write rep
                     (regfile, dest,
                      (alu (op, read (regfile, src1),
                                read (regfile, src2)))))
```

There is only one piece of state, the register file. The program counter is not visible since the pipeline does not perform instruction fetching. The operation is simple: if there is a stall, return the register file unchanged, otherwise return the register file with the destination register updated using the results from operating on the two source registers.

In the interest of brevity, we do not reproduce the HOL specification of the pipeline. The description is a state transition function on all of the state visible in Figure 4. The pipeline takes three cycles to complete an operation.

The abstraction function for the proof is generated from the pipeline specification. The abstraction function works by flushing the pipeline after a single instruction has been executed in order to force all of the results from that instruction through the pipeline. So, for the simple pipeline of Figure 4, the abstraction function is as follows:

```
⊢def abs (regfile, op1, op2, op_ex, dest_ex, bubble_ex,
          result, dest_wb, bubble_wb) =
     let ex1 = bubble_wb →  regfile
                          | write (regfile, dest_wb, result) in
     bubble_ex →  ex1
               | write (ex1, dest_ex, alu (op_ex, op1, op2))
```

The abstraction function takes the pipeline state as its argument and returns the top–level state (in this case, just the register file). The abstraction function can return four different results:

1. If both `bubble_wb` (indicating a pipeline bubble in the write back stage) and `bubble_ex` (indicating a pipeline bubble in the execute stage) are true, then the register file is returned unchanged.
2. If `bubble_ex` is true and `bubble_wb` is false, the register file is updated with the result of in the write back stage.
3. If `bubble_wb` is true and `bubble_ex` is false, the register file is updated with the result of the current ALU operation.
4. If both `bubble_wb` and `bubble_ex` are true, the register file is updated with both results.

The correctness result is a commuting relationship between the pipeline, `Imach`, and the top–level, `Smach`:

```
⊢def (abs  (Imach (regfile, op1, op2, op_ex, dest_ex, bubble_ex,
                  result, dest_wb, bubble_wb)
                 (stall, op, src1, src2, dest))) =
   Smach (abs (regfile, op1, op2, op_ex, dest_ex, bubble_ex,
               result, dest_wb, bubble_wb))
         (stall, op, src1, src2, dest)
```

The left–hand side of the equality executes the `Imach` once and then flushes it (using the `abs` function). The right–hand side flushes the pipeline and then executes the `Smach` on the resulting state once. If these are equal, the parallel `Imach` is said to correctly implement the sequential `Smach`.

At first, it appears that the flushing operation creates a situation where each instruction is verified independent of all the others. This is not the case. The pipeline is full of instructions when the pipeline is finished and all of these, along with their interactions and capacity for creating hazards, are completely executed. Thus, each instruction is verified in the presence of a pipeline full of instructions, and the verification proves that each instruction executes correctly regardless of what instructions are in the pipeline ahead of it.

Discussion. The correctness statement used by Burch is a relationship between the sequential machine and the parallel machine. The formulation and abstraction function are particularly well suited for verifications done using model checkers since no higher–order logic is required. Using higher–order logic, the specifications can be adapted to a predicate–based format that relates the state at time $t+1$ to the state at time t. Under such a formulation, the abstraction function becomes:

```
⊢def tabs (regfile, arg1, arg2, op_ex, dest_ex, bubble_ex,
          result, dest_wb, bubble_wb) t =
      regfile ((t+1)+1)
```

Recall that the abstraction function maps the lower–level state stream into the top–level state stream. The top–level state stream has only one component, `regfile`, which is a temporal projection of the lower–level register file (offset two time steps).

Using the abstraction function `tabs` we can reformulate the correctness statement to have the following form:

```
⊢ ∀ s e .
   (∀ t . (s(t+1)) = (Imach (s t) (e t)) ⇒
   (∀ t . (tabs s (t+1)) = Smach (tabs s t) (e t))
```

This correctness statement has the same format as the correctness statements of Section 3.3.

Certainly showing that one proof using the Burch method of formulating the correctness statement can be redone using the methodology of Section 3.3 does not constitute a proof that all such proofs could be so reformulated. Even so, after studying the Burch method and working with it over a period of months, we believe that this is true in general. More work is needed on the general structure of the two methods before a conclusive proof could be given.

While we are convinced that proofs done in the Burch methodology can be reformulated into the methodology of Section 3.3 we are not convinced that the opposite holds true. There is an inherent assumption in the Burch methodology that the pipeline results can be flushed in a reasonable number of steps. We are not convinced that the method will scale well to superscalar designs with pipelines that contain loops.

Indeed, the difficulty of the proof increases significantly as the number of stages that need to be flushed increases. As mentioned above, Burch does a proof of DLX as well as the simple pipeline presented here. We redid Burch's proof of DLX in HOL as well. The difference in the complexity is astounding. The proof of the simple pipeline presented here was done in 1.25 person hours (including transliterating the specifications) and runs in less than 4 seconds in HOL on an HP735. The proof of DLX using the Burch correctness statement took several days of proof time on an HP735 in HOL; Burch's model checker takes only 66 seconds to do the same proof. While the model checker does an admirable job of dealing with the case explosion, as the complexity of the pipeline increases the proof time will go up significantly.

7 The Uinta Correctness Statement

From the discussion of the last section, one might conclude that proving pipelines correct in HOL is hopeless for all but the most simple ones since even a pipeline like DLX takes several days of HOL proof time to complete. This section discusses the proof of a pipeline very similar to DLX and shows that it can be completed in reasonable time. Moreover, we show that the proof time is linear in the number of instructions and the number of pipeline stages.

Uinta has a five stage pipeline which presents data and control hazards (there are no structural hazards). Mitigation of the data hazards is done using two levels of data forwarding; mitigation of the control hazards is accomplished using a delayed branch (2 stages). Our verification makes no assumptions about software constraints or the ordering of instructions.

The instruction set semantics and pipeline specification for Uinta are too large to fit in the small amount of space available in this section. Interested readers are refered to longer discussions in [Win94, WC94]. The correctness statement depends on an abstraction function that maps the pipeline state into the instruction level state.

```
⊢def abs s t =
      let t′ = Temp_Abs (λ t. ¬(STALL (Ir (s t)) (Ir1 (s t)))) t in
      let j = (STALL (Ir (s (t′ + 1))) (Ir1 (s (t′ + 1)))) →  1 | 0 in
      let reg     = PipelineReg (s (t′ + 4)) and
          pc      = PipelinePc (s t′) and
          npc     = PipelinePc (s (t′+1)) and
          nnpc    = PipelinePc (s ((t′ + 2) + j)) and
          imem    = PipelineIMem (s t′) and
          dmem    = PipelineDMem (s (t′ + 3)) in
      State reg pc npc nnpc imem dmem
```

The `abs` function is a curried function of two arguments. The first argument is the pipeline level state stream; the second argument is the architectural level time. Thus, `abs ps` represents the architectural level state stream that is an abstraction of the pipeline level state stream `ps`.

In the function, the variable `t'` is defined using the temporal abstraction function `Temp_Abs` [Win95] as the next time after `t` when there is not a stall. The variable `j` gets the value 1 if the next instruction stalls and 0 otherwise.

An architectural level state record has five components: the register file, the program counter, the next program counter, the next next program counter, the instruction memory, and the data memory.

If we examine Figure 2, we see that, for example, the register file in the architectural model at time t, is the register file in the pipeline model skewed by 4 from the next time after t there is not a stall. Thus, we define `reg`, the register file component of the architectural level state record as:

```
      let reg     = PipelineReg (s (t′ + 4)) and
```

Other components of the architectural level state are defined similarly. Notice that the `pc`, `npc`, and `nnpc` are all defined in terms of the pipeline model program counter with differing skews. The value of `nnpc` is shifted when the next instruction stalls as indicated in Figure 2.

Using the abstraction function defined above, we can formulate a correctness statement for Uinta as follows:

```
⊢ Pipeline_Interp s e p ⇒  Arch_Interp (abs s) e p
```

The overall correctness theorem shows that the pipeline model implies the architectural model when it is applied to an abstraction of the pipeline state stream. This result can be combined with the proofs of the other levels (using the generic interpreter theory) to get a result that states the architectural model follows from the electronic block model.

Discussion. The correctness statement of Uinta follows the format of Section 3.3. Our goal in doing the proof of Uinta was to show that pipelined microprocessors could be done in the format traditionally used in non–pipelined microprocessor verifications. Not only has this format proven useful in proving properties about the architecture [Win91] and for extending the verification hierarchy [Win93], but we have developed a number of tools and techniques for dealing with specifications and correctness theorems in this format which we were loath to abandon.

The proof method used to verify Uinta is symbolic execution of the pipeline. The method is thus linear in the number of pipeline stages since as more pipeline stages are added, the time to perform symbolic execution increases linearly. Loops in the pipeline can be handled using loop invariants. The symbolic execution must be done for each instruction in the instruction set, so adding new instructions increases the proof time linearly. The overall proof time for the correctness statement is thus $O(n_s i)$ where n_s is the number of pipeline stages and i is the number of instructions. The proof of Uinta runs in HOL in less than two hours on an HP735.

8 Conclusions

In this paper we have proposed a format for the correctness statement in microprocessor verification. The format is the same whether the microprocessor is pipelined or not; only the style of the abstraction function need change. The format we propose has two advantages:

1. The result clearly states that the parallel machine represented by the pipeline behaves in the same manner as the sequential machine represented by the instruction set semantics. This is intuitively what a user of a microprocessor expects.
2. The result is useful in follow–on work that builds on the verification result to either prove properties of the microprocessor or in which the microprocessor is but one component of a larger system being verified.

We have shown four examples of verified pipelines and discussed where they differed from the format proposed here. Our intent is not to call into question the work of these researchers; each had goals other than simply verifying a pipeline. Rather, our goal was to point out how different formulations vary and to compare them to our correctness statement.

References

[BD94] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In *Proceedings of the 1994 Conference on Computer–Aided Verification*, June 1994.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5, 1940.

[GG91] Stephen J. Garland and John V. Guttag. A guide to lp, the larch prover. Technical Report 82, digital Systems Research Center, December 1991.

[GM93] Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.

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

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

[SHGG93] James B. Saxe, James J. Horning, John V. Guttag, and Stephen J. Garland. Using transformations and verification in circuit design. *Formal Methods in System Design*, 3(3), December 1993.

[TK93] Sofiene Tahar and Ramayya Kumar. Implementing a methodology for formally verifying RISC processors in HOL. In Jeffery J. Joyce and Carl Seger, editors, *Proceedings of the 1993 International Workshop on the HOL Theorem Prover and its Applications*, August 1993.

[WC94] Phillip J. Windley and Michael Coe. The formal verification of instruction pipelines. In Ramayya Kumar and Thomas Kropf, editors, *Proceedings of the 1994 Conference on Theorem Provers in Circuit Design*, Lecture Notes in Computer Science. Springer–Verlag, September 1994.

[Win90] Phillip J. Windley. A hierarchical methodology for the verification of microprogrammed microprocessors. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1990.

[Win91] Phillip J. Windley. Using correctness results to verify behavioral properties of microprocessors. In *Proceedings of the IEEE Computer Assurance Conference*, June 1991.

[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. Specifying instruction set architectures in HOL: A primer. In Thomas Melham and Juanito Camilleri, editors, *Proceedings of the 1994 International Workshop on the HOL Theorem Prover and its Applications.* Sspringer Verlag, Spetember 1994.

[Win95] Phillip J. Windley. Formal modeling and verification of microprocessors. *IEEE Transactions on Computers*, 44(1), January 1995.