A Correctness Model for Pipelined Microprocessors

Phillip J. Windley[1] and Michael L. Coe[2]

[1] Laboratory for Applied Logic, Brigham Young University,Provo, UT 84602-6576
[2] Laboratory for Applied Logic, University of Idaho, Moscow, ID 84843-1010

Abstract. What does it mean for an instruction pipeline to be correct? We recently completed the specification and verification of a pipelined microprocessor called UINTA. Our proof makes no simplifying assumptions about data and control hazards. This paper presents the specification, describes the verification, and discusses the effect of pipelining on the correctness model. The most significant effect on the pipeline is that data and temporal abstractions in the correctness model are *not* orthogonal as they are in non–pipelined implementations.

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. See [Gor83, Bow87, CCLO88, Coh88, Joy88, Hun89, Win90, Her92, SWL93, Win94a] for examples.

The verification of pipelined microprocessors presents unique challenges. The correctness model is somewhat different than the standard correctness models used previously (see Section 7.1). Besides the correctness model, the concurrent operations inherent in a pipeline lead to *hazards* which must be considered in the proof. There are three types of hazards:

- **structural hazards** which arise because of resource constraints (i.e. more than one operation needing the ALU at a time),
- **data hazards** which arise when data is needed before it has been calculated or, alternately when data is changed before it has been used, and
- **control hazards** which arise when instructions change the flow of control after some operations in the original flow of control have already begun.

Several papers have presented the verification of pipelined microprocessors: In [SB90], the verification of a three stage pipelined machine name Mini-Cayuga is presented. The verification is the first, to our knowledge, of a pipelined microprocessor. Because the pipeline has only three stages, however, the verification did not have to deal with data and control hazards in the pipeline.

The verification of a machine similar to the DLX processor of [HP90] is presented in [TK93]. The machine has a five stage pipeline and encounters data and control hazards, but it is not clear from the presentation whether these are dealt with in the proof or in the assumptions.

This paper presents the verification of a pipelined microprocessor called UINTA. 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.

Our work in microprocessor verification has been characterized by the development of formal models for microprocessor correctness and a standard model of microprocessor semantics [Win93]. In [Win94a] we present the verification of a non–pipelined microprocessor using our model, which we call the generic interpreter theory. The generic interpreter theory does several things:

1. The formalization provides a step–by–step approach to microprocessor specification by enumerating the important definitions that need to be made for any microprocessor specification.
2. Using the formalization, the verification tool can derive the lemmas that need to be verified from the specification.
3. After these lemmas have been established, the verification tool can use the formalization to automatically derive the final result from the lemmas.

Using the generic interpreter theory provides a standardized model that ensures that the theorems used can be put together in standard ways and used in other places in the proof. One of the goals of the effort presented here was to evaluate the use of the generic interpreter theory in verifying pipelined processors. We will see that while the generic interpreter theory provides the same benefits for most of the verification of UINTA, its fails in one important place. This is discussed in more detail in Section 7.1.

The specification and verification of UINTA is done hierarchically to reduce the abstraction distance between successive layers. As noted in [Mel88], there are four types of abstraction: structural, behavioral, data, and temporal. Where possible, we limit the types of abstraction between any two layers. The four specification models employed in the verification are:

- **Electronic Block Model.** This model is a structural description of register transfer level. The model states how the major components such as the register file and arithmetic logic unit (ALU) are connected together.
- **Phase Model.** This model is a behavioral abstraction of the electronic block model. There is no data or temporal abstraction between the electronic block model and the phase model.
- **Pipeline Model.** This model is a temporal abstraction of the phase model. The two phases of the phase model are combined in the pipeline model. Each time unit in the pipeline model represents one execution of each stage in the pipeline.
- **Architectural Model.** This model is a data and temporal abstraction of the pipeline model. the architectural model describes the instruction set semantics and is intended to represent the assembly language programmer's

view of the microprocessor. We will say more about why we perform the data and temporal abstract concurrently in Section 7.1.

The verification of UINTA shows that the resultant specifications and theorems need not be different from those used in non–pipelined microprocessor verification, but that the correctness model and the important lemmas change considerably. We will briefly present the specifications of each level (in a slightly different order than that above) and concentrate on the parts of the verification that differ significantly from previous microprocessor verifications.

2 A Brief Introduction to HOL

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`." 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.
- Comments in HOL are enclosed in percent signs, `%`

3 Architectural Specification

Our intent is to present just enough of the specification of the architectural level to show that it is unchanged from the standard model and to support the discussion of the verification. Our presentation follows that of any denotational semantics: we discuss the syntax, the semantic domain, and the denotations, in that order. We conclude by showing the specification developed from the denotations using the generic interpreter theory. A more complete discussion of the use of HOL for specifying architectures is available in [Win94b].

3.1 Instruction Set Syntax

The instruction set for UINTA contains 27 instructions. The small number is not an issue since, as we show later, the verification would not change significantly with the addition of new instructions and the proof time is $O(n)$ in the size of the instruction set.

The instruction set contains instructions from most of the important classes of instructions one would find in any instruction set: ALU instructions, immediate instructions, branch instructions, jump instructions, load instructions, and store instructions. The following is the abstract syntax for part of the instruction set:

```
Instruction =
        LDI   *ri *ri *short    |
        STI   *ri *ri *short    |
        ADD   *ri *ri *ri       |
        ADDI  *ri *ri *short    |
        JMP   *word26           |
        BEQ   *ri *short        |
        NOOP                    |
        ...
```

3.2 Semantic Domain

The semantic domain is a record containing the state variables that the assembly language programmer would see. The name of the record and the name of each field is given in backquotes and the type of each field is enclosed in double quotes:

```
create_record `State`
   [`Reg`, ":*ri->*wordn";      % register file %
    `Pc`,  ":*wordn";           % program counter %
    `NPc`,  ":*wordn";          % next program counter %
    `NNPc`, ":*wordn";          % next next program counter %
    `Imem`, ":*memory";         % instruction memory %
    `Dmem`, ":*memory";         % data memory %
   ];;
```

The register file is modeled as a function from register indices to n–bit words, the program counters are n-bit words. **Imem** and **Dmem** are both memories. The legal operations on n-bit words and memories are specified algebraically. We do not present those specifications here. Interested readers are referred to [Win94b].

The three instances of the program counter in the semantic domain are an artifact of the delayed branches. Because delayed branches appear to the assembly language programmer, they are visible at the architectural level. We will see that in lower level of the specification hierarchy, there is only one program counter and the three program counters of the architectural level are merely temporal projections of the single program counter.

The separation of the memory into instruction and data memory is a convenience that allows us to ignore self modifying programs. Self modifying programs do not cause much concern in a non-pipelined machine, but when instructions are pipelined, an instruction in the pipeline can modify another instruction that has already been loaded and started to execute. This kind of behavior hardly seems worth the trouble it causes, so we disallow it.

3.3 Instruction Denotations

Instruction denotation can be given for classes of instructions. We call these specifications *semantic frameworks* since they specify a framework for the semantics of an entire class of instructions. They are similar to the class level specifications of [TK93]. For example, here is the semantic framework for the ALU instructions in UINTA. Notice that it is parameterized by the ALU operation to be performed, op:

```
⊢def ALU_FM op Rd Ra Rb s e =
      let reg      = Reg s and
          pc       = Pc s and
          nextpc   = NPc s and
          nextnextpc  = NNPc s and
          imem      = Imem s and
          dmem      = Dmem s in
      let a        = INDEX_REG Ra reg and
          b        = INDEX_REG Rb reg in
      let result   = op (a, b) in
      let new_reg = UPDATE_REG Rd reg result and
          new_pc = nextpc and
          new_nextpc = nextnextpc and
          new_nextnextpc  = inc nextnextpc in
      (State new_reg new_pc new_nextpc new_nextnextpc imem dmem)
```

The framework is also parameterized by the destination register index, **Rd** and the source register indices, **Ra** and **Rb**. Because the function is curried, applying **ALU_FM** to an operation and the register indices like so:

```
(ALU_FM add Rd Ra Rb)
```

returns a state transition function (i.e., a function that takes a state, s, and environment. e, and returns a new state).

Lines 2–7 of the preceding definition bind local names to the contents of the fields of the state s. Lines 8–9 bind a and b to the contents of the register file, reg, at indices Ra and Rb respectively. The op parameter is used to calculate the result in line 10. Lines 11–14 calculate new values for those members of the state that change in this framework. For example, in line 11, a new register file is calculated by updating the old register file at location Rd with the result calculated in line 10. Line 15 creates the new state record that is returned as the result of the function.

We create a denotation for the instruction set by relating the instruction syntax to the semantic frameworks using the following definition:

```
⊢def (M_INST (LDI Rd Ra imm) =
              LOAD_FM Rd Ra imm)              ∧
     (M_INST (ADD Rd Ra Rb) =
              ALU_FM add Rd Ra Rb)            ∧
     (M_INST (SUB Rd Ra Rb) =
              ALU_FM sub Rd Ra Rb)            ∧
     (M_INST (ADDI Rd Ra imm) =
              ALUI_FM add Rd Ra imm)          ∧
     (M_INST (BNOT Rd Ra) =
              UNARY_FM bnot Rd Ra)            ∧
     (M_INST (JALI Rd imm) =
              JALI_FM Rd imm)                 ∧
     (M_INST (BEQ Ra imm) =
              BRA_FM eqzp Ra imm)             ∧
     ...
```

M_INST maps a valid instruction, given syntactically, to a state transition function denoting the meaning of that instruction.

3.4 Interpreter Specification

The architectural level specification is created by the generic interpreter theory from the preceding definitions:

```
⊢ Arch_Interp s e =
        (∀t.
          let k = Opcode s e in
          (s (t + 1)) = M_INST k (s t) (e t))
```

The definition, in classic form, declares that the state of the architecture, s, at time $t+1$ is a function, M_INST, of the state at time t.

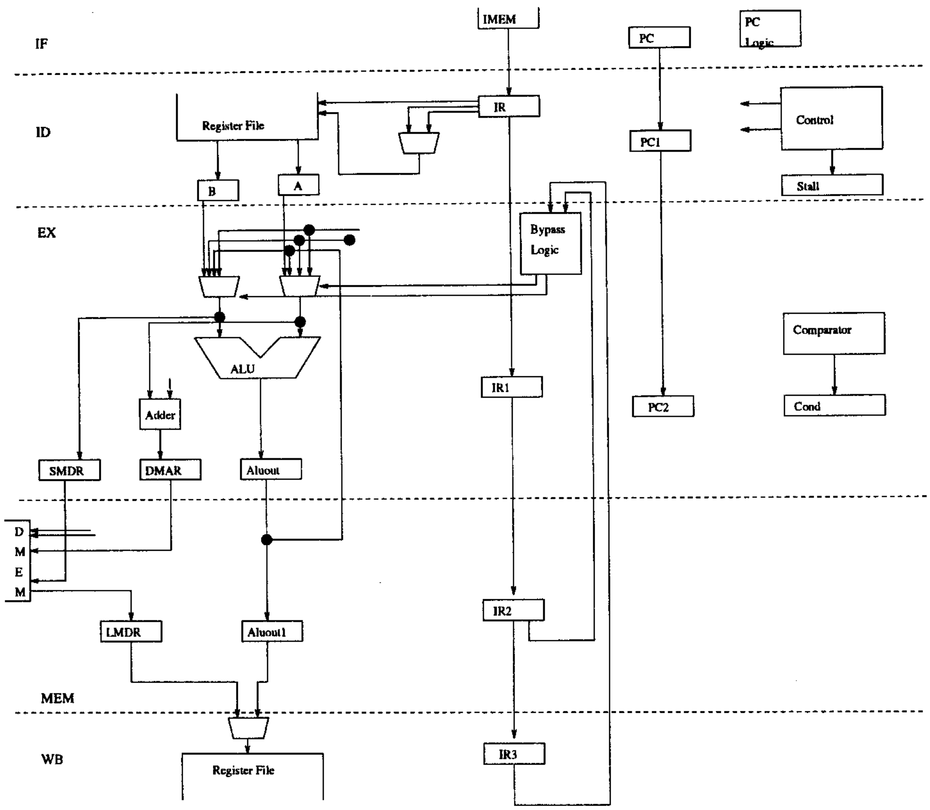

Fig. 1. UINTA Electronic Block Model

4 Electronic Block Model

The electronic block model, EBM, is a structural model of the register transfer level and is shown in Figure 1. The model describes the connections between the major components of the microprocessor. The EBM is the lowest level in the verification hierarchy. For the most part there is a recognizable correspondence between the EBM and synthesizable statements in a hardware description language such as VHDL.

The EBM state record is shown in Figure The state of the EBM is, obviously, larger than the state of the architectural level. Comparing the state record with Figure 1 shows that the EBM state record contains a field for each register and flipflop in the implementation. The EBM state record contains a field for each component of the architectural state record as well as all of the state invisible at the architectural level. Note that the next program counter, **npc**, and next next program counter, **nnpc** are not present in the electronic block model; we will discuss the disappearance of these later. The stage markers (in the comments) indicate the stage in which the register is set, not the stage in which it is used.

```
create_record `EbmState`
   [`EbmReg`,      ":*ri->*wordn";      % register file %
    `EbmPc`,       ":*wordn";           % program counter %
    `EbmIMem`,     ":*memory";          % instruction memory %
    `EbmDMem`,     ":*memory";          % data memory %
    `EbmIr`,       ":*wordn";           % instruction register, fetch %
    `EbmIr1`,      ":*wordn";           % instruction register, decode %
    `EbmIr2`,      ":*wordn";           % instruction register, execute %
    `EbmIr3`,      ":*wordn";           % instruction register, memory %
    `EbmA`,        ":*wordn";           % ALU input latch A %
    `EbmB`,        ":*wordn";           % ALU input latch B %
    `EbmPc1`,      ":*wordn";           % program counter, decode %
    `EbmPc2`,      ":*wordn";           % program counter, execute %
    `EbmALUout`,   ":*wordn";           % ALU output latch %
    `EbmALUout1`,  ":*wordn";           % ALU output latch, memory  %
    `EbmDMAR`,     ":*wordn";           % data memory address register %
    `EbmSMDR`,     ":*wordn";           % store memory data register %
    `EbmLMDR`,     ":*wordn";           % load memory data register %
    `EbmCond`,     ":bool";             % branch condition flipflop %
    `EbmStall`,    ":bool";             % stall flipflop %
    `clk`,         ":bool";             % 2 phase clock %
   ];;
```

Fig. 2. UINTA Electronic Block Model State Record

The top–level description of the EBM connects three large blocks; the control block, the clock, and the data path; together. The structure is modeled in the usual existentially quantified conjunction of predicates format. Each of the predicates is itself an existentially quantified conjunction of predicates. When fully expanded, the structural definition of UINTA is approximately four pages of text.

```
⊢def uintaEBM s e p  =
     ∃ (clk_1 clk_2 wrsig rsig newstall:time->bool).
    (CONTROL_BLOCK (EbmIr o s, EbmIr1 o s, EbmIr2 o s,
                    clk_1, EbmStall o s, newstall, rsig, wrsig))  ∧
    (CLOCK_SPEC (clk o s, clk_1, clk_2 ))  ∧
    (DATA_PATH  (EbmReg o s, EbmPc o s, EbmIMem o s,
         EbmDMem o s, EbmIr o s, EbmIr1 o s, EbmIr2 o s, EbmIr3 o s,
         EbmA o s, EbmB o s, EbmPc1 o s, EbmPc2 o s, EbmALUout o s,
         EbmALUout1 o s, EbmDMAR o s, EbmSMDR o s, EbmLMDR o s,
         EbmCond o s, EbmStall o s, clk o s, clk_1, clk_2,
         wrsig ,rsig, newstall))
```

The arguments to the predicates are not just values, but signals (time dependent values). Thus, most of the arguments to the predicates are some function name, such as **EbmDMem**, composed with the EBM state stream **s**. Because the state stream, **s**, is a function with type "**:time →EbmState**", composing a field selector function with **s** returns a function with type "**:time →f**" where **f** is the type of the field. For example, **EbmDMem o s** has type "**:time →*memory**".

5 Phase Model Specification

The phase model provides a behavioral abstraction of the EBM. The behavior of the phase model is equivalent to the EBM (i.e. there is not data or temporal abstraction). The phase model can be specified and verified from the EBM using the generic interpreter theory.

We do not give the details of the specification or verification here. Like the verification of most behavioral abstractions, the proof is quite irregular, but not technically difficult.

6 Pipeline Model Specification

The pipeline model is a temporal abstraction of the phase model. The behavior of the two phases is collapsed into one behavior for the pipeline. The data abstraction that takes place is mostly related to the temporal abstraction. Thus, the state description is largely the same for the pipeline model as for the EBM.

The specification of the state transition in the pipeline model is given as one large function since all of the changes take place concurrently. We will present the behavior by stages, but keep in mind that the stages make their state changes concurrently.

Fetch Stage. The primary state change in the fetch stage is loading the instruction register:

```
% stalls occur when ... %
let stall = STALL ir ir1 in
% fetch stage %
let new_ir = stall →  ir
                    | decode_word (
                          fetch (imem, address pc)) in
```

The instruction register is unchanged in the event of a stall and gets the current instruction from memory otherwise.

Decode Stage. The primary state change in the decode stage is loading the ALU input latches, **A** and **B** from the register file:

```
% decode stage, use ir and pc %
let new_ir1 = stall →  NOOP
                     | ir and
    new_pc1 = stall →  pc1 | pc and
    new_a = INDEX_REG (sel_Ra ir) new_reg and
    new_b = ((CLASSIFY ir) = STORE) →
                   INDEX_REG (sel_Rd ir) new_reg |
                   INDEX_REG (sel_Rb ir) new_reg in
```

In the event of a stall, the decode stage instruction register, **ir1** gets a **NOOP** rather than the instruction in **ir**.

Execute Stage. The execute stage can be broken into two large blocks. The first block describes the state changes associated with the ALU:

```
% execute stage, use ir1 and pc1 %
 1  let bsrc = ((CLASSIFY ir1) = STORE) →  (sel_Rd ir1) |
 2                                        (sel_Rb ir1) in
 3  let ra = (sel_Ra ir1) and
 4      imm = (sel_Imm ir1) in
 5  let amux = 5
 6       (¬(ra = R0)) →
 7         (((ra = (sel_Rd ir2)) ∧
 8          (CLASSIFY ir2 = JUMPLINK)) →  pc2     |
 9         ((ra = (sel_Rd ir2)) ∧
10          (IS_REG_WRITE ir2))       →  aluout  |
11         ((ra = (sel_Rd ir3)) ∧
12          (IS_REG_WRITE ir3))       →  aluout1 |
13         ((ra = (sel_Rd ir3)) ∧
14          (CLASSIFY ir3 = LOAD))    →  lmdr    |
15                                       a)      |
16                                       a       and
17      bmux = ...   in
18  let new_ir2 = ir1 and
19      new_pc2 = pc1 and
20      new_dmar = add (amux, shl (sel_Imm ir1, 2)) and
21      new_smdr = bmux and
22      new_aluout =
23        ((CLASSIFY ir1) = ALU)      →  ((BINOP ir1) (amux, bmux)) |
24        ((CLASSIFY ir1) = ALUI)     →  ((BINOP ir1) (amux, imm)) |
25        ((CLASSIFY ir1) = UNARY)    →  ((UNOP ir1) amux) |
26                                       ARB in
```

The new value of the **aluout** latch is calculated from the value given by the **amux** and **bmux**. These multiplexors supply the correct value for the respective inputs to the ALU based on the data forwarding conditions. For example, in lines 9–10, if the destination of the previous instruction in the pipe is the same as the **A** source for the current instruction and that instruction writes to the register file, then we supply the value in **aluout** to the **A** input to the ALU rather than the value in the **A** latch.

The second major state change in the execute stage is the calculation of the new value for the program counter:

```
% execute stage, use ir1 and pc1 %
 1  let cond = (CMPOP ir1 amux) in
 2  let new_pc = stall                                → pc |
 3              (((CLASSIFY ir1) = BRA) ∧ cond) →
 4                      (add (pc2, shl (sel_Imm ir1, 2)))    |
 5              ((CLASSIFY ir1) = JUMPLONG)       →
 6                      (add (pc2, shl (sel_Imm26 ir1, 2))) |
 7              ((CLASSIFY ir1) = JUMPREG)        →
 8                      (add (amux, shl (sel_Imm ir1, 2)))   |
 9              ((CLASSIFY ir1) = JUMPLINK)       →
10                      (add (pc2, shl (sel_Imm ir1, 2)))    |
11                   (inc pc)  in
```

Note that in the event of a stall, the program counter does not change (line 2). The new value of the program counter if the current instruction is a branch and the condition is true, for example, is the sum of **pc2** and the immediate portion of the instruction shifted left twice for word boundary alignment (lines 3-4).

Memory Stage. The memory stage calculates values for the load memory data register if the current instruction is a load or the stores a value in the data memory if the current instruction is a store:

```
let new_ir3 = ir2 and
    new_lmdr = ((CLASSIFY ir2) = LOAD) →
                    (fetch (dmem, address dmar)) |
                    lmdr and
    new_dmem = ((CLASSIFY ir2) = STORE) →
                    (store (dmem, address dmar, smdr)) |
                    dmem and
    new_aluout1 =((CLASSIFY ir2) = JUMPLINK) →  pc2 |
                                                aluout in
```

Write Back Stage. The write-back stage updates the register file if necessary:

```
let new_reg =
    (IS_REG_WRITE ir3)    →  UPDATE_REG (sel_Rd ir3) reg aluout1 |
    (CLASSIFY ir3 = LOAD) →  UPDATE_REG (sel_Rd ir3) reg lmdr |
                          reg in
```

The New Pipeline State. The new values calculated in the preceding code fragments comprise the updated pipeline state. The state returned from the pipeline model is a new pipeline state record created be the following expression:

```
(PipelineState new_reg new_pc imem new_dmem
               new_ir new_ir1 new_ir2 new_ir3
               new_a new_b new_pc1 new_pc2
               new_aluout new_aluout1
               new_dmar new_smdr new_lmdr)
```

Verifying the Pipeline Correct. The proof that the pipeline model is correctly implemented by the phase model is done within the generic interpreter theory. There are no special considerations or exceptions.

7 Verifying UINTA

In this section we will concentrate on the proof that the pipeline model implements the architectural model since this is the part of the proof that differs from previous microprocessor verifications. We first discuss why the correctness model at this level in the proof differs from our previous notion of correctness and then discuss the proof itself in three crucial areas: important lemmas, the proof tactic, and the correctness theorem.

7.1 The Correctness Model

The correctness model presented by the generic interpreter theory is based a notion of state stream abstraction. In the model, state stream u is an abstraction of state stream u' (written $u \preceq u'$) if and only if

1. each member of the range of u is a state abstraction of some member of the range of u' and
2. there is a temporal mapping from time in u to time in u'.

There are two distinct kinds of abstraction going on: the first is a data abstraction and the second is a temporal abstraction. Thus, using a state abstraction function, $\mathcal{S}$, and a temporal abstraction function, $\mathcal{F}$, we define stream abstraction as follows

$$u \preceq u' \equiv \exists(\mathcal{S} : \mathbf{S}' \rightarrow \mathbf{S}).\ \exists(\mathcal{F} : \mathbf{N} \rightarrow \mathbf{N}).\ \mathcal{S} \circ u' \circ \mathcal{F} = u$$

where $\circ$ denotes function composition. The important part of this model for our purposes is the orthogonality of the temporal and data abstractions (indicated by function composition).

This is the crux of the problem with using the generic interpreter theory for verifying pipelined microprocessors: the data and temporal abstractions in

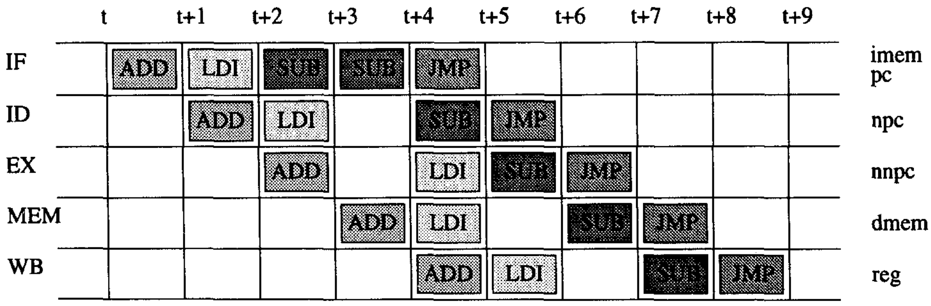

Fig. 3. Uinta Pipeline Execution

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 3. 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 3:

- 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. Indeed, the architectural state variables, **pc**, **npc**, and **nnpc** are all the same variable in the pipeline model, they are merely different temporal views of the same data.

7.2 State Stream Abstraction

The discussion in the last section suggests that a function mapping the state stream of a pipeline into a state stream for a non–pipelined model must collect different pieces of the pipelined state stream at different times and package them into a state record to appear in the non-pipelined state stream at a particular time. This section describes such an abstraction function for UINTA.

The correctness model for UINTA depends on a function, called **abs** that maps a pipeline model state stream into an architectural model state stream. This function allows us to maintain the illusion in the architectural model that the state changes occur in a single time step between times t and $t+1$.

```
⊢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 abstract of the pipeline level state stream **ps**.

In the function, the variable `t'` is defined using the temporal abstraction function `Temp_Abs` [Win94a] 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 3.2: 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 3, 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 3.

7.3 Important Lemmas

There are two important lemmas in the verification of UINTA. The first, the data forwarding lemma, proves that the data forwarding behavior of the pipeline is correct:

```
 1 ⊢ Pipeline_Interp
 2     (λ t. PipelineState (reg t) (pc t) (imem t) (dmem t) (ir t)
 3                          (ir1 t) (ir2 t) (ir3 t) (a t) (b t) (pc1 t)
 4                          (pc2 t) (aluout t) (aluout1 t) (dmar t)
 5                          (smdr t) (lmdr t))
 6    (λ t. Env(ivec t)(int t)(reset t)) (λ t. p t) ⇒
 7  ∀ t.
 8  let new_reg =
 9        INDEX_REG x1
10              (IS_REG_WRITE(ir3(t+1)) →
11                 UPDATE_REG (sel_Rd(ir3(t+1)))
12                            (reg(t+1))
13                            (aluout1(t+1)) |
14              (CLASSIFY(ir3(t+1)) = LOAD) →
15                 UPDATE_REG (sel_Rd(ir3(t+1)))
16                            (reg(t+1))
17                            (lmdr(t+1)) |
18             reg(t+1)) in
19   ¬((CLASSIFY(ir1(t + 1)) = LOAD) ∧  (x1 = sel_Rd(ir1(t + 1)))) ⇒
20   ((INDEX_REG x1(reg(t+4))) =
21   ((¬(x1 = R0)) →   (((x1 = sel_Rd(ir2(t+2))) ∧
22                       (CLASSIFY(ir2(t+2))=JUMPLINK)) →  pc2(t+2) |
23                      ((x1 = sel_Rd(ir2(t+2))) ∧
24                       IS_REG_WRITE(ir2(t+2)))      →  aluout(t+2) |
25                      ((x1 = sel_Rd(ir3(t+2))) ∧
26                       IS_REG_WRITE(ir3(t+2)))      →  aluout1(t+2) |
27                      ((x1 = sel_Rd(ir3(t+2))) ∧
28                       (CLASSIFY(ir3(t+2))=LOAD)) →  lmdr(t+2) |
29                       new_reg |
30                      new_reg))
```

The theorem states that the pipeline model (lines 1–6) implies that reading, **x1**, from the register file at time $t + 4$ (line 20) is equivalent to the data forwarding behavior of the pipeline (lines 21–30) which is reading values at time $t + 2$.

The other important lemma is used to limit the case analysis in the proof. In the proof that any given instruction works, we want to be very general about the instruction sequencing. Thus we do not want to make any assumptions about whether the instructions before the current instruction stalls or not. This means that as we do the proof, we have to do a case analysis on stalling. Whenever we don't stall, we can proceed with the the symbolic execution of the instruction and complete the proof for that case, but there is always the left–over stall case to consider. The following lemma, however, limits the proof to one stall case split by showing that the pipeline cannot stall twice in a row (because the stall inserts **NOOP** which cannot cause a stall):

```
⊢ Pipeline_Interp
    (λ t. PipelineState (reg t) (pc t) (imem t) (dmem t) (ir t)
                         (ir1 t) (ir2 t) (ir3 t) (a t) (b t) (pc1 t)
                         (pc2 t) (aluout t) (aluout1 t) (dmar t)
                         (smdr t) (lmdr t))
    (λ t. Env(ivec t)(int t)(reset t)) (λ t. p t) ⇒
 (∀t . (STALL (ir t) (ir1 t)) ⇒
      ¬(STALL (ir (t+1)) (ir1 (t+1))))
```

Without this lemma, the symbolic execution would lead to an infinite number of case splits on stalling. Every pipeline, to be correct, must have a similar lemma showing that it cannot issue an unlimited number of stalls. If it could, of course, the processor could go into an infinite loop of sorts under certain conditions. This is not usually the behavior one wants from a processor.

7.4 The Instruction Tactic

The proof of UINTA breaks into 27 cases, one for each of the instructions in the instruction set. [3] Each of these instructions can be solved using the same tactic. The tactic considers each instruction under the case that it stalls and the case that is does not. In either case, the method of proof is symbolic execution of the pipeline model using a general purpose symbolic execution tactic that we developed for microprocessor proofs.

A single /uinta/ instruction can be verified in less than 10 minutes on an HP 735 running HOL88 version 2.01 compiled with AKCL. Increasing the size of the instruction set thus increases the overall verification time linearly. Unless the instruction differs significantly from the instructions already in the instruction set, it is likely that the tactic used to perform the verification will not change, so the human effort to add additional instructions is probably small.

When there is an error, either in the design or the specification, the tactic fails, leaving a symbolic record of what the pipeline computed in each stage. We

[3] For larger instruction sets, we could have done case analysis on instruction classes (which correspond to the semantic frameworks) to limit the case explosion.

have found that this record is very helpful for debugging the implementation and specification as it is usually quite easy to see where what the pipeline computed differs from what the designer expected.

7.5 The Correctness Theorem

The overall correctness theorem (for this level in the proof hierarchy) shows that the pipeline model implies the architectural model when it is applied to an abstraction of the pipeline state stream:

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

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 EBM. The overall form of the goal is familiar and has not changed from the goals commonly used in non-pipelined verification.

8 The Correctness Model and Hazards

The correctness model we have developed handles read after write (RAW) data hazards in the general case as shown in the data forwarding lemma of Section 7.3. The same cannot be said of control and structural hazards because of the variety of design techniques for mitigating them. RAW data hazards are almost always mitigated by forwarding the needed information from later stages of the pipeline to earlier stages. Because a single technique suffices, a single model suffices.

UINTA mitigates control hazards using delayed branching. As the discussion in Section 3 shows, delayed branching is visible at the architectural level. Another popular technique for mitigating control hazards, branch prediction, would *not* be visible at the architectural level. We have verified a processor that uses a simple form of branch prediction. As expected, changing the architectural model to that extent has significant effects on the abstraction function. For now, as the techniques for mitigating control hazards change, so will our verification methodology.

While we have not used our methodology for reasoning about a pipelined architecture with structural hazards, we believe that the techniques we have outlined are sufficient with some minor changes. The most important change relates to skew in the abstraction function: the abstraction function of Section 7.2 uses a specific number for the skew. A structural hazard could make the skew non-deterministic. In that case, the skew would have to be determined from a signal indicating when the structural hazard had cleared in the same way that asynchronous memory is specified now (see [Win94a] for more information). Aagaard and Leeser have described a very general methodology for reasoning about pipelines with structural hazards [AL94].

9 Conclusion

We have completed the specification and verification of a pipelined microprocessor called UINTA. UINTA has a five stage pipeline with two levels of data forwarding and delayed branching. This paper has presented techniques for verifying that the pipeline model correctly implements the architectural model.

Because we have developed tools for dealing with specifications and correctness theorems in a standard format, one of our goals in the verification of UINTA was to ensure that the specification and correctness theorem were stated using that format. The standard format has proven useful in proving properties about the architecture [Win91] and for extending the verification hierarchy [Win93]. The verification in this paper was completed using specifications exactly like those used for non-pipelined microprocessors. The correctness result is exactly like the correctness results for non-pipelined microprocessors.

We have also presented the abstraction function at the heart of the correctness model and shown how and why it differs from the abstractions commonly used to verify non-pipelined microprocessors. This function is the essence of what it means to say that the UINTA pipeline correctly implements the UINTA architecture. The function allows us to preserve the illusion that instructions execute sequentially in the architectural model even though the pipelined implementation performs operations in parallel.

We are presently verifying a pipelined microprocessor called SAWTOOTH. SAWTOOTH has a different set of features that UINTA including a simple form of branch prediction, user and multi–level system interrupts, a windowed register file, and supervisory mode. We plan to use the experience of verifying UINTA and SAWTOOTH to modify the generic interpreter theory to include pipeline semantics.

References

[AL94] Mark D. Aagaard and Miriam E. Leeser. Reasoning about pipelines with structural hazards. In Ramayya Kumar and Thomas Kropf, editors, *Proceedings of the 1994 Conference on Theorem Provers in Circuit Design*. Springer–Verlag, September 1994.

[Bow87] Jonathan P. Bowen. Formal specificaiton and documentation of microprocessor instruction sets. In *Microprocessing and Microprogramming 21*, pages 223–230, 1987.

[CCLO88] S. D. Crocker, E. Cohen, S. Landauer, and H. Orman. Reverification of a microprocessor. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 166–176, April 1988.

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

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

[Coh88] Avra Cohn. A proof of correctness of the VIPER microprocessor: The first level. In G. Birtwhistle and P. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis,* pages 27–72. Kluwer Academic Publishers, 1988.

[Gor83] Michael J.C. Gordon. Proving a computer correct. Technical Report 41, Computer Lab, University of Cambridge, 1983.

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

[Her92] John Herbert. Incremental design and formal verification of microcoded microprocessors. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Theorem Provers in Circuit Design, Proceedings of the IFIP WG 10.2 International Working Conference, Nijmegen, The Netherlands.* North-Holland, June 1992.

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

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

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

[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.

[SB90] M. Srivas and M. Bickford. Formal verification of a pipelined microprocessor. *IEEE Software,* 7(5):52–64, September 1990.

[SWL93] E. Thomas Schubert, Phillip J. Windley, and Karl Levitt. Report on the ucd microcoded viper verification project. In Jeffery J. Joyce and Carl Seger, editors, *Proceedings of the 1993 International Workshop on the HOL Theorem Prover and its Applications.,* August 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.

[Win90] Phillip J. Windley., *The Formal Verification of Generic Interpreters.* PhD thesis, University of California, Davis, Division of Computer Science, June 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.

[Win94a] Phillip J. Windley. Formal modeling and verification of microprocessors. *IEEE Transactions on Computers,* 1994. (to appear).

[Win94b] 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.