A Scalable Formal Verification Methodology for Pipelined Microprocessors

Jeremy Levitt and Kunle Olukotun
Computer Systems Laboratory, Stanford University, CA 94305

Abstract

We describe a novel, formal verification technique for proving the correctness of a pipelined microprocessor that focuses specifically on pipeline control logic. We iteratively deconstruct a pipeline by merging adjacent pipeline stages, allowing for the verification to be done in several easier steps. We present an inductive proof methodology that verifies that pipeline behaviour is preserved as the pipeline depth is reduced via deconstruction; this inductive approach is less sensitive to pipeline depth and complexity than previous approaches. Invariants are used to simplify the proof, and datapath components are abstracted using validity checking with uninterpreted functions. We present experimental results from the formal verification of a DLX five-stage pipeline using our technique.

1 Introduction

As the complexity of modern microprocessors increases they become more and more difficult to design correctly. Yet, design errors can be very expensive to fix. The cost of design errors increases as the design progresses; errors found after a processor has been fabricated can cost millions of dollars to fix. Thus, techniques that can detect bugs in the early stages of design are critical.

Extensive simulation has long been used to detect errors during the design process. However, due to the exponential increase in the complexity of modern microprocessors, simulation has become a design bottleneck. A substantial portion of the design cycle is now devoted to simulation, and yet simulation provides only partial coverage; design errors can and do occasionally slip through. Designs are increasingly limited by the time required to simulate them with any degree of confidence.

Formal methods [9] offer the promise of being able to verify that a design is correct (or detect where bugs exist) and doing it in a fraction of the time it would take to exhaustively simulate the same design [10]. Current formal techniques fall into two classes: automatic methods and theorem proving techniques. Automatic methods which explore a state graph include CTL model checking [3] and state exploration systems such as Murphi [5]. Theorem proving methods are based on general theorem proving systems such as the HOL system [6].

While the desired behavior of a microprocessor may be concisely expressed by its instruction set architecture (ISA), the implementation is often greatly complicated by efforts to maximize performance. Features introduced purely to increase performance, such as pipelining, register renaming and superscalar execution, exponentially increase the complexity of the final design. Pipelining in particular is a major source of complexity. A significant amount of extra state is added to the model in order to execute different instructions simultaneously. And complex control logic is required to cope with pipeline hazards and maintain the programmer's illusion of sequential execution [7].

Previous attempts to apply formal methods to the verification of pipelined microprocessors have not scaled well with pipeline complexity. Theorem proving methods have been successfully applied in [4][11]. However, these examples all required large amounts of expert user time and verified only simple pipelines. Automatic methods have also been used to verify pipelined processors. In [2] a method is presented for verifying the control logic of a pipelined microprocessor using a validity checker for a logic of uninterpreted functions with equality. In [1] a method is presented for verifying pipelined processors that checks that a specific kind of relation holds between the implementation and the specification. While both of these methods are a significant improvement over previous techniques, they do not promise to scale well to longer, more complex pipelines.

In this paper we present an automatic formal verification technique specifically developed to tackle the complexity due to pipelining. Our technique, which we call *unpipelining*, removes pipeline stages from an implementation while preserving the implementation's behavior, collapsing it into a single stage through a series of transformations. The complexity due to pipelining is completely eliminated and the deconstructed pipeline can be compared directly to the ISA specification. Unpipelining builds on previous work [8] by using equality checking with uninterpreted functions to abstract the datapath.

A pipeline deconstruction transformation merges the two deepest pipeline stages. After merging the stages we prove using an inductive argument on the number of execution cycles that the behaviour of the implementation is preserved. The inductive argument allows the direct comparison of the two very similar *before* and *after* pipelines and breaks the verification task into smaller components which are easier to verify. By proving that the behaviour is preserved after every transformation, we can guarantee that the final unpipelined implementation has the same behavior as the original pipelined implementation. If the proof fails, we can produce a test vector that exposes the bug in the pipeline.

The rest of this paper is organized as follows. In Section 2 we define correctness and introduce pipeline implementation techniques covered by unpipelining. In Section 3 we describe the unpipelining transformation and in Section 4 we introduce the proof methodology and present an example. Section 6 and Section 7 present experimental results and conclusions.

2 Preliminaries

2.1 Correctness

Formal verification consists of comparing a processor *implementation* against a *specification* and proving that behaviour of the implementation meets that prescribed by the specification. The specification is an ISA-level model that specifies some processor state (SPEC) and how the state is modified by the sequential execution of instructions. Processor state may consist of a program counter (PC), a register file (RF), an instruction cache (I-cache) and a data cache (D-cache). Since instructions execute sequen-

33rd Design Automation Conference ®

DAC 96 - 06/96 Las Vegas, NV, USA

tially, data and control dependencies between instructions executed by a specification are communicated through the processor state. For example, an instruction which uses the result of a previous instruction would fetch it from the RF.

A pipelined processor *implementation* is much more complex. It contains extra logic to maintain the programmer's illusion of sequential instruction execution, while in fact executing several instructions in parallel. It also has additional state in order to represent the different instructions executing in parallel.

The specification does not precisely prescribe the behaviour of an implementation. Variables that appear in an implementation and not in the specification are not constrained at all. Variables appearing in both may not match exactly on a cycle to cycle basis. For example, the specification may state that an instruction updates the register file and the program counter in the same cycle, while in a particular implementation an instruction may update the register file several cycles after updating the program counter.

An implementation is defined to be correct with respect to a specification if the specification and the implementation produce exactly the same results for every program. At the conclusion of any program the memory contents produced by an implementation and its specification must be equivalent. If there is a bug in the implementation then some sequence of instructions exists that cause the implementation and the specification to finish execution with different memory states. Using this definition of correctness, an implementation can be correct even if the implementation state never exactly matches the specification state during program execution.

2.2 Pipelining

Pipelines divide the execution of an instruction into a number of steps. Each step is known as a pipeline stage. The depth of a pipeline corresponds to the number of stages in the pipeline. Instructions enter a pipeline at stage one and, under normal operation, progress to the next stage on each new clock cycle. Once an instruction has progressed through all the pipeline stages, its execution is completed.

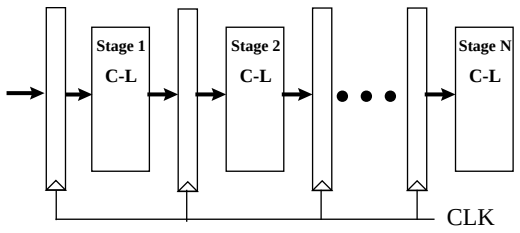

Figure 1. Pipelines execute instructions in stages.

Pipelining is complicated by pipeline hazards. Different instructions are in different steps of execution in each pipeline stage. Hazards arise when an instruction executing in an earlier pipeline stage requires the result of an instruction executing in a later pipeline stage. There are three basic techniques for coping with pipeline hazards: bypassing, squashing and stalling [7]. Bypassing forwards values produced by an instruction deep in the pipeline to an instruction earlier in the pipeline. Squashing annuls the execution of instructions earlier in the pipeline. Stalling delays the execution of instructions earlier in the pipeline.

Bypassing, squashing and stalling all require the addition of feedback logic to forward results from later pipeline stages to earlier pipeline stages. This feedback can be built from the three network topologies shown in Figure 2. Bypassing uses type (a); data is forwarded from a deeper stage *j* to an earlier stage *i* based on the comparison of the instructions in each stage. Squashing uses type (c); an instruction earlier in the pipeline is nullified based on a signal determined by the results of an instruction deeper in the pipeline. Stalling uses both types (b) and (c); type (c) is used to insert a bubble into the pipeline and type (b) is used to delay the execution of earlier stages by one cycle, the control signals being determined from a combination of instructions in the pipeline.

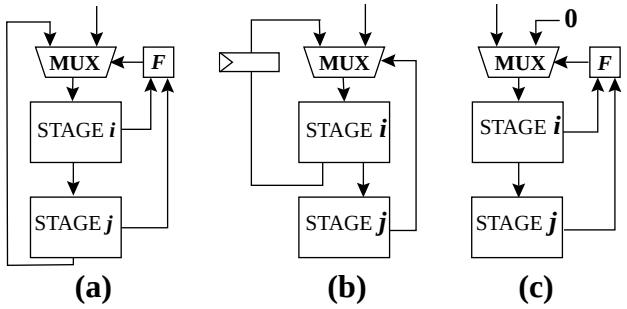

Figure 2. Pipeline feedback blocks.

Figure 3 shows an example of a pipeline in execution. Here, instruction i_1 stalls the instructions behind it in the pipeline. Pipeline stages one through five have been labeled IF (instruction fetch), ID (instruction decode), EX (execute), MEM (memory) and WB (register write-back) respectively, corresponding to the five stages of instruction execution presented in [7].

Cycle	1	2	3	4	5	6	7	8
Instr i_1	IF	ID	EX	MEM	WB			
Instr i_2		IF	ID	stall	EX	WB	MEM	
Instr i_3			IF	stall	ID	EX	WB	MEM

Figure 3. A pipeline in execution.

3 Pipeline Deconstruction

Pipeline deconstruction is the process of merging pipeline stages. If there were no pipeline hazards, then merging pipeline stages would amount to simply removing the latches between them. The logic that copes with hazards introduces feedback between stages; this makes pipeline deconstruction more difficult.

An iteration of pipelining deconstruction merges the deepest two pipeline stages. Feedback between the deepest pipeline stage and earlier stages is removed depending on the type of pipeline hazard. In the case of stalling, both the logic that inserts the bubble into the pipeline and the logic that stalls subsequent instructions are removed. For squashing, only the logic that squashes instructions in the second to last stage is removed; logic that squashes instructions in other stages is unaffected as it is still required.

Since instructions complete one clock cycle earlier after an iteration of unpipelining, all processor state that was updated in the deepest stage will be updated one cycle earlier. This will eliminate certain hazards associated with that stage and eliminate the need for the feedback logic to deal with them, making it safe to remove the feedback. Conversely, if feedback logic were not removed, it would become unclocked feedback when the latches between the pipeline stages involved are removed by subsequent iterations of pipeline deconstruction.

Pipeline feedback can be represented using the topologies shown in Figure 2. Feedback is removed by determining which multiplexer input will be selected in the absence of a pipeline hazard and connecting that input directly to the multiplexer output. Once the feedback has been removed, the latches between the two stages can be removed. Figure 4 shows how instructions flow though a pipeline with stages EX, MEM and WB combined. Notice that the stalling between instructions i_1 and i_2 has been eliminated.

Cycle	1	2	3	4	5
Instr i_1	IF	ID	EX/MEM/WB		
Instr i_2		IF	ID	EX/MEM/WB	
Instr i_3			IF	ID	EX/MEM/WB

Figure 4. A partially unpipelined pipeline in execution.

Pipeline deconstruction is a mechanical process that yields a correctly functioning pipeline on each iteration assuming the original pipeline implementation is correct. The process of determining whether or not an implementation is correct is described in the next section.

4 Verification of Equivalence

After an iteration of deconstructing pipeline P, the resulting pipeline P' must preserve P's behaviour. We prove the equivalence of P and P' using an inductive argument on the number of execution cycles. For each processor state variable and feedback variable in P, we form an inductive hypothesis expressing the variable in terms of variables in P'.

By definition, instructions executed on P' must produce the same results as on P to be correct. The hazards in P' are the same as those that were not eliminated by unpipelining from P. Since the hazards are the same and the results must be the same, the same values must be forwarded whenever a hazard is detected. This forms the basis for the induction hypotheses.

While we need only show that these hypotheses hold to show correct operation, if we assume generally that the same values are forwarded even when hazards are not detected, the proof can be further simplified. At worst, the more general assumption results in a false negative; the assumption can then be tightened to exclude this case. However, we have not required this in practice.

4.1 Bypassing

To understand how the induction hypotheses are generated, consider first the case where only bypassing logic has been removed. As seen in Figure 5 and Figure 6, instructions executing on P' complete execution a cycle earlier than their counterparts on P, since the latch between the deepest two stages has been removed. Thus, processor state variables and feedback values generated in the deepest stage should be produced one cycle earlier by P'. Processor state variables and feedback values generated in any other stage should be unaffected by the iteration of unpipelining and should match cycle for cycle.

Cycle	1	2	3	4	5	6
Instr i_1	IF	ID	EX	MEM	WB	
Instr i_2		IF	ID	EX	MEM	WB

Figure 5. Pipeline P

Cycle	1	2	3	4	5
Instr i_1	IF	ID	EX	MEM/WB	
Instr i_2		IF	ID	EX	MEM/WB

Figure 6. Pipeline P'

Definition 1: *Let $T(v)$ be the stage in which a processor state variable or feedback value v is assigned.*

Let n be the number of stages in pipeline P. For each processor state variable or feedback value v the inductive hypotheses are:

$$(T(v) < n) \Rightarrow (v'@i \equiv v@i) \quad \text{(EQ 1)}$$

$$(T(v) \equiv n) \Rightarrow (v'@i \equiv v@(i+1)) \quad \text{(EQ 2)}$$

The $@$ is used to indicate the cycle of execution at which the variable is evaluated. For example, $RF@42$ would be the evaluation of the register file at clock cycle 42. Primed variables belong to P' whereas unprimed variables belong to P.

For each inductive hypothesis, the base case is true by construction of P'. Thus, one need only show that assuming all the hypotheses hold at cycles $i < k$, then they hold at cycle k. This is done by forming left and right-hand-side equations. The left hand side equation expresses $v'@k$; and the right hand side expresses $v@(k+1)$ or $v@k$ depending on the stage value of $T(v)$. Using the next-state equations for P', $v'@k$ is expanded. Whenever a variable for which there is an inductive hypothesis appears in the expansion it is replaced and the expansion for that variable stops. Since there are inductive hypotheses for all the feedback variables, this process is guaranteed to eventually terminate. The left hand side will then be expressed entirely in terms of variables from P. In lock step, the right hand side is also expanded using the next-state equations for P.

Validity checking with uninterpreted functions is used to compare the left and right hand sides, and determine if they are equivalent. The validity checker does not know about the next-state equations and will not perform any additional expansion on the terms; all necessary expansion is performed before calling the validity checker. To deal with the bypassing logic that has been removed, the term that replaced the bypassing multiplexer in P' must be expanded until a $READ(WRITE(\ldots))$ combination is encountered. As described in [8], the validity checker interprets $READ$ and $WRITE$ operations and automatically reasons that:

$$READ(WRITE(RF, addr1, value), addr2) \Leftrightarrow MUX2(addr1 \equiv addr2, value, READ(RF, addr2)) \quad \text{(EQ 3)}$$

where $MUX2(control, input1, input2)$ is a two input multiplexer.

4.2 Stalling and Squashing

Squashing and stalling disrupt the flow of instructions in the pipeline. By removing such logic we complicate the relationship between P' and P. Let P be a three-stage pipeline that uses a branch-not-taken strategy with a two cycle penalty for misprediction. Figure 7 and Figure 8 show P and P' in execution when the branch is taken.

Because the final pipeline stages in P were merged and the squashing logic removed, P' never squashes i_3. P' calculates the correct PC one cycle earlier and avoids fetching i_3.

Stalling is similar. Consider a pipeline with a load-use stall between stages 2 and 3, as shown in Figure 9. Since the result of the load instruction is not ready until the MEM/WB stage, the EX

Cycle	1	2	3	4	5	6
Branch	IF	ID	EX/MEM/WB			
Instr i_2		IF	squash			
Instr i_3			squash			
Instr i_4				IF	ID	EX/MEM/WB

Figure 7. P executing a branch instruction.

Cycle	1	2	3	4
Branch	IF	ID/EX/MEM/WB		
Instr i_2		squash		
Instr i_4			IF	ID/EX/MEM/WB

Figure 8. P' executing a branch instruction.

stage of i_2 must stall. In Figure 10 we see that P' performs the load one cycle earlier than P so that the result can be forwarded to i_2 without any stalling.

Cycle	1	2	3	4	5	6
Load	IF	ID	EX	MEM/WB		
Instr i_2		IF	ID	stall	EX	MEM/WB

Figure 9. P executing a load instruction.

Cycle	1	2	3	4
Load	IF	ID	EX/MEM/WB	
Instr i_2		IF	ID	EX/MEM/WB

Figure 10. P' executing a load instruction.

Since pipeline deconstruction removes stalling and squashing logic, P and P' become increasingly unsynchronized with every stall or squash that occurs in P and not in P'. To synchronize the pipelines, we introduce induction variables t_i and s_i. We define s_i inductively as $s_o \equiv 0$, $squash@(i+1+s_i+t_i) \Rightarrow (s_{i+1} \equiv s_i + 1)$ and $\neg squash@(i+1+s_i+t_i) \Rightarrow (s_{i+1} \equiv s_i)$. $squash$ is a signal automatically extracted from P that is asserted whenever a squash occurs in P that does not occur in P'. t_i is defined similarly for stalls.

Intuitively, s_i is the number of clock cycles that P and P' are out of synchronization due to squashes when P' is at cycle i, and t_i is the number of clock cycles P and P' are out of synchronization due to stalls. Using s_i and t_i to synchronize the two pipelines and letting $k \equiv T(v)$ and n be the depth of pipeline P, the induction hypotheses are:

$$k < n \Rightarrow v'@i \equiv v@(i + s_{i+n-k-1} + t_i) \quad \text{(EQ 4)}$$

$$k \equiv n \Rightarrow v'@i \equiv v@(i+1+s_i+t_i) \quad \text{(EQ 5)}$$

Using these hypotheses, the technique of generating and expanding left and right hand side equations can be applied. However, a few more assertions must be provided to the validity checker to enable it to reason successfully that the two sides are equivalent. Since stalls and squashes nullify the previous pipeline stage, two squashes or stalls cannot occur in immediate succession. Nullified instructions do not squash other instructions or stall the pipeline. Thus, using the previous definition we can assert:

$$\neg squash@(i+s_i+t_i) \quad \text{(EQ 6)}$$

$$\neg stall@(i+s_i+t_i) \quad . \quad \text{(EQ 7)}$$

Nullified instructions furthermore should not modify processor state. (All of these assertions regarding nullified instructions are easily checked in P.) This provides the additional assertion:

$$\begin{aligned}&(squash@(i+1+s_i+t_i) \vee stall@(i+1+s_i+t_i)) \Rightarrow \\ &\quad \forall (v \in SPEC) \\ &\qquad T(v) \equiv n \Rightarrow (v@(i+1+s_i+t_i) \equiv v@(i+2+s_i+t_i)) \wedge \\ &\qquad T(v) \equiv n-1 \Rightarrow (v@(i+s_i+t_i) \equiv v@(i+1+s_i+t_i))\end{aligned} \quad \text{(EQ 8)}$$

5 Example

Here we present an example of applying unpipelining to a simple four-stage pipeline. We focus on the key aspects of unpipelining, showing how the pipeline is transformed after each iteration and sketching out the inductive part of the verification proof.

The pipeline, shown in Figure 11 uses a predict-not-taken branch prediction scheme. If the prediction is wrong, squashing logic between stages two and three nullifies the mispredicted instruction. Bypassing logic between stages three and four resolves data hazards.

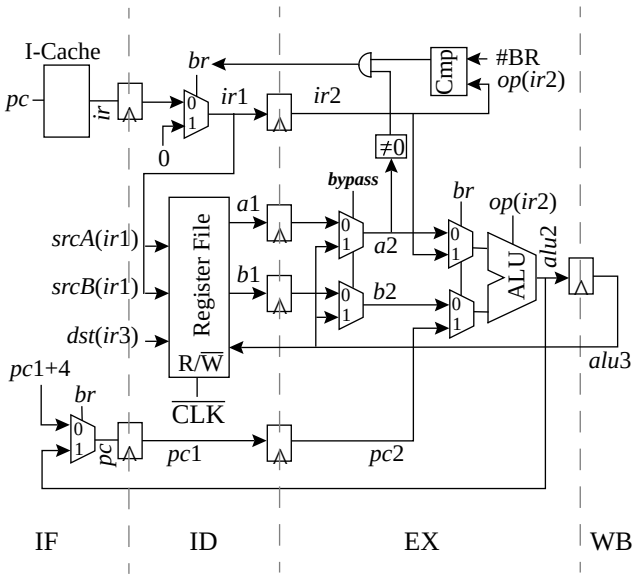

Figure 11. Schematic for 4-stage pipeline.

The state machine representation for the processor is given in Figure 12. n (new) variables are next-state variables and o (old) variables are current-state variables. By definition, $v_n@i \equiv v_o@(i+1)$. Wires a and b have been introduced to make the description simpler. The state equations are automatically classified by pipeline stage. This classification is based on the number of cycles for a change in the PC to propagate to the left hand side of a state equation.

The first unpipelining iteration merges stages 3 and 4 in P and removes the bypassing logic between them to produce P'. P' is shown in Figure 13; the changes made to remove the bypassing and the latch between stages 3 and 4 are emphasized in bold text.

To verify that P' and P have the same behaviour we generate the induction hypotheses using (EQ 1) and (EQ 2):

stage	next state equations
1	$pc_n = MUX2(br_n, alu2_n, ADD(pc_o, 4))$
2	$ir1_n = MUX2(br_n, 0, READ(icache, pc_o))$ $pc1_n = pc_o$ $a1_n = READ(rf_n, srcA(ir1_n))$ $b1_n = READ(rf_n, srcB(ir1_n))$
3	$ir2_n = ir1_o$ $pc2_n = pc1_o$ $a2 = MUX2(dst(ir3_n) \equiv srcA(ir2_n), alu3_n, a1_o)$ $b2 = MUX2(dst(ir3_n) \equiv srcB(ir2_n), alu3_n, b1_o)$ $br_n = (op(ir2_n) \equiv \#BR \wedge a2 \neq 0)$ $alu2_n = MUX2(br_n, ALU(+, pc2_n, imm(ir2_n)),$ $ALU(func(ir2_n), a2, b2))$
4	$ir3_n = ir2_o$ $alu3_n = alu2_o$ $rf_n = WRITE(rf_o, dst(ir3_n), alu3_n)$

Figure 12. Original description, P.

stage	next state equations
1	$pc_n = MUX2(br_n, alu2_n, ADD(pc_o, 4))$
2	$ir1_n = MUX2(br_n, 0, READ(icache, pc_o))$ $pc1_n = pc_o$ $a1_n = READ(rf_n, srcA(ir1_n))$ $b1_n = READ(rf_n, srcB(ir1_{n'}))$
3	$ir2_n = ir1_o$ $pc2_n = pc1_o$ $a2 = \mathbf{a1}_o$ $b2 = \mathbf{b1}_o$ $br_n = (op(ir2_n) \equiv \#BR \wedge a2 \neq 0)$ $alu2_n = MUX2(br_n, ALU(+, pc2_n, imm(ir2_n)),$ $ALU(func(ir2_n), a2, b2))$ $ir3_n = \mathbf{ir2}_n$ $alu3_n = \mathbf{alu2}_n$ $rf_n = WRITE(rf_o, dst(ir3_n), alu3_n)$

Figure 13. After one iteration of unpipelining, P'.

(1) $rf'@i = rf@(i+1)$

(2) $pc'@i = pc@i$

(3) $alu3'@i = alu3@(i+1)$

(4) $alu2'@i = alu2@i$

(5) $br'@i = br@i$.

The last three induction hypotheses are required due to feedback paths in P.

We assume that all the induction hypotheses hold $@i; \forall (i < k)$ and show they hold $@k$. The derivation for $br'@k = br@k$ is given in Figure 14. Here the left and right sides are expanded and the induction hypotheses are used to get both expressions in terms of variables in P. Since bypassing logic has been removed, additional expansion is performed on the right-hand side and (EQ 3) is applied. For brevity "… " has been used to replace terms that have already been shown to be equivalent. The other induction hypotheses can be similarly verified.

LHS Equation

$br'@k$

$= op(ir1'@k{-}1) \equiv \#BR \wedge a1'@k{-}1 \neq 0$

$= op(MUX2(br'@k{-}1, 0, READ(icache', pc'@k{-}2)) \equiv \#BR$
$\wedge READ(rf'@k{-}1, srcA(ir1'@k{-}1)) \neq 0$

$= op(MUX2(br@k{-}1, 0, READ(icache, pc@k{-}2)) \equiv \#BR$
$\wedge READ(rf@k, srcA(ir1@k{-}1)) \neq 0$

$= \ldots \wedge READ(WRITE(rf@k{-}1, dst(ir3@k), alu2@k{-}1),$
$srcA(ir1@k{-}1)) \neq 0$

$= \ldots \wedge MUX2(dst(ir3@k) \equiv srcA(ir1@k{-}1), alu2@k{-}1,$
$READ(rf@k{-}1, srcA(ir1@k{-}1))) \neq 0$

RHS Equation

$br@k$

$= op(ir1@k{-}1) \equiv \#BR \wedge$
$MUX2(dst(ir3@k) \equiv srcA(ir1@k{-}1), alu3@k, a1@k{-}1) \neq 0$

$= op(MUX2(br@k{-}1, 0, READ(icache, pc@k{-}2)) \equiv \#BR \wedge$
$MUX2(dst(ir3@k) \equiv srcA(ir1@k{-}1), alu2@k{-}1,$
$READ(rf@k{-}1, srcA(ir1@k{-}1))) \neq 0$

Figure 14. Proof of inductive hypothesis for br'

Next we apply the pipeline deconstruction transformation again on P'. P'' shown in Figure 15 is produced by merging stages 2 and 3 and removing the squashing logic between them. Changes are in bold text.

stage	next state equations
1	$pc_n = MUX2(br_n, alu2_n, ADD(pc_o, 4))$
2	$ir1_n = READ(icache, pc_o)$ $pc1_n = pc_{o'}$ $a1_n = READ(\mathbf{rf}_o, srcA(ir1_n))$ $b1_n = READ(\mathbf{rf}_o, srcB(ir1_n))$ $ir2_n = \mathbf{ir1}_n$ $pc2_n = \mathbf{pc1}_n$ $a2 = \mathbf{a1}_n$ $b2 = \mathbf{b1}_n$ $br_n = (op(ir2_n) \equiv \#BR \wedge a2 \neq 0)$ $alu2_n = MUX2(br_n, ALU(+, pc2_n, imm(ir2_n)),$ $ALU(func(ir2_n), a2, b2))$ $ir3_n = ir2_n$ $alu3_n = alu2_n$ $rf_n = WRITE(rf_o, dst(ir3_n), alu3_n)$

Figure 15. After two iterations of unpipelining, P''

Because squashing logic has been removed, (EQ 4) and (EQ 5) are used to generate the induction hypotheses. They are:

(1) $pc''@i \equiv pc'@(i + s_{i+1})$

(2) $rf''@i \equiv rf'@(i + s_i + 1)$

(3) $alu2''@i \equiv alu2'@(i + s_i + 1)$

(4) $br''@i \equiv br'@(i + s_i + 1)$

Since the feedback path through $alu3$ has been eliminated in P', we now have only two additional induction hypotheses. The details of the proof are similar to the previous iteration, except for

the use of the induction variable s_i. Figure 16 shows the proof of the induction hypothesis for br''. Note that br' is a squash signal, allowing us to simplify the right hand side using (EQ 6). $rf'@(k+s_k)$ is reduced to $rf'@(k+s_{k-1})$ using (EQ 8).

LHS Equation

$$
\begin{aligned}
& br''@k \\
&= op(ir1''@k) \equiv \#\mathrm{BR} \wedge a1''@k \neq 0 \\
&= op(READ(icache'', pc''@k{-}1) \equiv \#\mathrm{BR} \\
&\qquad \wedge READ(rf''@k{-}1, srcA(\ldots)) \neq 0 \\
&= op(READ(icache', pc'@(k-1+s_k)) \equiv \#\mathrm{BR} \\
&\qquad \wedge READ(rf'@(k+s_{k-1}), srcA(\ldots)) \neq 0
\end{aligned}
$$

RHS Equation

$$
\begin{aligned}
& br'@(k+s_k+1) \\
&= op(ir1'@(k+s_k)) \equiv \#\mathrm{BR} \wedge a1'@(k+s_k) \neq 0 \\
&= op(MUX2(br'@(k+s_k), 0, \\
&\qquad READ(icache', pc'@(k-1+s_k)))) \equiv \#\mathrm{BR} \\
&\qquad \wedge READ(rf'@(k+s_k), srcA(\ldots)) \neq 0 \\
&= op(READ(icache', pc'@(k-1+s_k))) \equiv \#\mathrm{BR} \\
&\qquad \wedge READ(rf'@(k+s_{k-1}), srcA(\ldots)) \neq 0
\end{aligned}
$$

Figure 16. Proof of inductive hypothesis for br''

The final step in the proof is to use the validity checker to compare P'' directly with the ISA description.

6 Results

We have experimented on a variety of simple pipelines, the most advanced being the five-stage DLX pipeline with a predict-not-taken strategy for branches, a load-use interlock [7] and six operation classes: ALU immediate, 3-register ALU, branches, jumps, loads and stores.

Starting with a state machine description of the processor, the pipeline deconstruction was performed automatically. For every unpipelining iteration the appropriate inductive hypotheses were determined automatically. Expansion required some manual intervention to overcome shortcomings in our software. The inductive hypotheses were then input to a validity checker for verification.

For the most complex DLX pipeline, pipeline deconstruction and the generation of inductive hypotheses took 2.8s on a 200MHz SGI Indy. Table 1 shows the time required to verify the inductive hypotheses on an SGI Indy for each iteration.

stages merged	time
MEM - WB	2.1s
EX - WB/MEM	2.8s
ID - WB/MEM/EX	0.9s
IF - WB/MEM/EX/ID	0.9s

Table 1: Time required to verify induction hypotheses for DLX

Notice that the time required at each step is insensitive to the depth of the pipeline. Rather, the time is a factor of the number of induction hypotheses and the amount of feedback removed on that iteration of unpipelining.

7 Conclusions

We have presented a verification methodology that specifically targets the pipeline control logic of a microprocessor. It builds on previous work by using validity checking with uninterpreted functions. By introducing two novel ideas: pipeline deconstruction, and an inductive proof on the number of execution cycles, our technique is much better able to cope with pipeline complexity. We have implemented our algorithms in software and proven the correctness of a five stage DLX pipeline.

While unpipelining is not directly applicable to microprocessor implementation techniques such as register renaming or out-of-order execution, we have demonstrated the potential benefits of investigating domain specific verification. Future work will focus on developing domain specific verification techniques for other advanced microprocessor implementation techniques.

Acknowledgments

We would like to thank the reviewers for their helpful and insightful comments. This research was supported by ARPA contract ARMY DABT63-95-C-0049-P00001.

References

[1] V. Bhagwati and S. Devadas, “Automatic Verification of Pipelined Microprocessors,” in *Proceedings of 31st Design Automation Conference*, San Diego CA, June, 1994.

[2] J. R. Burch and D. L. Dill, “Automatic Verification of Pipelined Microprocessor Control,” in *Conference on Computer-Aided Verification*, Lecture Notes in Computer Science, vol 18, pp 68–80, Springer-Verlag, 1994.

[3] E. M. Clarke, E. A. Emerson and A. P. Sistla, “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic,” *ACM Transactions on Programming Languages and Systems*, vol 8, no. 2, April, 1986.

[4] A. J. Cohn, “A Proof of Correctness of the Viper Microprocessor: The First Level,” in *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P. A. Subrahmanyam, editors, pp. 27-72. Kluwer Academic Publishers, 1988.

[5] D. L. Dill, *et. al*, “Protocol verification as a hardware design aid,” in *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 522-525, Cambridge, MA, October, 1992.

[6] M. J. C. Gordon, “HOL: A proof generation system for higher-order logic,” in *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P. A. Subrahmanyam, editors, pp. 73–120. Kluwer Academic publishers, Boston, MA, 1988

[7] J. Hennessy and D. Paterson, *Computer Architecture: A Quantitative Approach, 2nd ed*. Morgan Kaufman, 1996.

[8] R. B. Jones, D. L. Dill, J. R. Burch, “Efficient Validity Checking for Processor Verification,” in *Proceedings of The International Conference on Computer Aided Design*, San Jose CA, Nov, 1995.

[9] M. C. McFarland, “Formal Verification of Sequential Hardware: A Tutorial,” *IEEE Transactions on computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 5, May 1993.

[10] K. McMillan, “Fitting Formal Methods into the Design Cycle,” in *Proceedings of 31st Design Automation Conference*, San Diego CA, June, 1994.

[11] M. Srivas and M. Bickford, “Formal Verification of a Pipelined Microprocessor” *IEEE Software*, vol. 7, no. 5), pp. 52–64, Sept. 1990