

Verifying Correct Pipeline Implementation for Microprocessors

Jeremy Levitt and Kunle Olukotun

Computer Systems Laboratory, Stanford, CA 94305

Abstract

We introduce a general, automatic verification technique for pipelined designs. The technique is based on a scalable, formal methodology for analysing pipelines. The key advantages to our technique are: it specifically targets pipeline control, making it more efficient; it requires no explicit specification, since it compares hardware against itself; it can be used within the broader framework of hierarchical verification; and, it can be easily extended to handle certain “complex” pipelined structures.

1 Introduction

Even with improved verification techniques, formal verification of entire systems such as a microprocessor design is likely to remain an intractable problem. However, by taking advantage of a design’s hierarchical structure and by partitioning the design into functional units, the correctness of many important design components can be formally verified. And, at a certain level of abstraction, designers can verify that these components interact correctly with one another.

The application of formal verification techniques can greatly increase a designer’s confidence in their design. Formal verification techniques explore the state space of a design very efficiently, allowing them to provide a level of coverage that is much greater than that practically achievable through either simulation or emulation. Formal verification techniques can also be applied early on in the design cycle, enabling them to expose bugs earlier which will reduce design costs and time-to-market.

To be capable of verifying a particular component in the design hierarchy, formal verification techniques must be able to abstract both the component’s interactions with other components and the behaviour of the component’s sub-components.

Formal verification techniques must also address the difficulty in determining correctness criteria for design components. To verify that a component is correctly implemented, some specification of the component’s correct behaviour is required. However, while the correct behaviour may be specified for an aggregate system, it is often not specified for arbitrary design components. The components in a design are, after all, the result of design-

ers attempting to meet overall system specifications. They are part of the implementation; they are not part of the original specifications.

In this paper we present a formal verification technique based on the methodology introduced in [5] for verifying the correctness of microprocessor pipelines. We extend this methodology, called *unpipelining*, making it more general, more automatic and easier to use. The key benefits of unpipelining are: it verifies a design against itself, thus removing the need for an external specification; it can be used within the framework of hierarchical verification; and it is scalable to longer, more complex pipelines.

Unpipelining builds on previous work [7] [3] by using equality checking with interpreted and uninterpreted functions to abstract hardware at lower levels of the hierarchy and to abstract the interface with other design components.

2 General Approach

Unpipelining differs from other techniques for verifying pipelined designs [1][2][4][10] in that it uses domain specific knowledge about pipeline implementation techniques to analyze the pipeline structure of a design. Using the results of this analysis, unpipelining automatically reverse engineers a pipeline through a series of transformations called *pipeline deconstruction*. Each application of pipeline deconstruction shortens the pipeline by merging its last two stages into one single stage.

To verify that a given pipeline has been correctly implemented, we prove that the behaviour of the pipeline is preserved after each deconstruction transformation. Any bugs in the pipeline’s implementation will be exposed, since they will cause the proof to fail for some iteration of pipeline deconstruction. If all the deconstruction transformations are successful, then not only is the pipeline verified, but it has been transformed into an functionally equivalent unpipelined design. This equivalent design can then be used at a higher level of the verification hierarchy as a simple abstraction for the pipelined component.

The proof requirements are automatically constructed using knowledge of the pipeline and of the deconstruction transformation most recently applied. An automatic decision procedure for determining the validity of expressions containing interpreted and uninterpreted functions with

equality [7] [3] determines whether or not the proof requirements are satisfied. The use of a decision procedure for uninterpreted functions allows for lower levels of the design hierarchy that are not relevant to the proof to be easily abstracted. The ability of the decision procedure to handle interpreted functions allows for the use of an abstract interface to other components in the system that interact with the pipelined component.

The rest of this paper is organized as follows. Section 3 introduces standard pipeline implementation techniques and defines what it means for a pipeline to be correctly implemented. Section 4 discusses pipeline analysis and the associated notion of a pipeline template. Section 5 explains the process of pipeline deconstruction. Section 6 goes through the proof processes. Section 7 presents experimental results; Section 8 details the limitations of the unpipelining method; and Section 9 presents conclusions.

3 Pipeline Model

Pipelining is an implementation technique that increases instruction execution rate. A pipeline divides the execution of an instruction into a number of steps. Each step is known as a pipeline stage. The number of stages in the pipeline is called the pipeline *depth*.

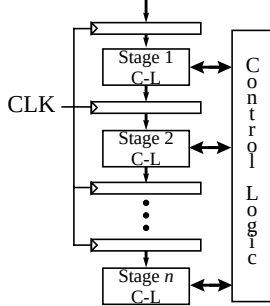


Figure 1. A Pipelined Circuit

We model a pipeline as a set of next-state equations. Non-control logic, particularly the datapath component, is abstracted using uninterpreted functions. The contents of memory elements such as caches and register files are accessed and modified explicitly using interpreted read and write functions. This abstraction separates verification of the pipeline from verification of the memory subsystems.

State-elements in the support set for the next-state equations fall into one of two categories: external-state and pipeline-state. External-state includes programmer visible state such as register files and the program counter, and it includes state supplied by units separate from the pipeline such as cached values and interrupts signals.

Pipeline-state is made up of the state held in the latches between pipeline stages.

The program counter (PC) is a specific element of external-state that designates the start of the pipeline. The first stage in executing an instruction to determine its PC; the next-state equation for the PC belongs in the first pipeline stage. Based on the number of cycles it takes a for a change in the PC to propagate to the left hand side of a next-state equation, the remaining next-state equations are partitioned into pipeline stages.

(EQ 1) is an example of a next-state equation for a PC, where *ex* is asserted when an exception is raised and *br* is asserted when a branch is to be taken. Variables with subscript *n* are next-state variables and those with subscript *c* are current-state variables.

$$pc_n = MUX2(ex_n, ex-pc_n, MUX2(br_n, br-pc_n, pc_c + 4)) \quad (EQ\ 1)$$

In a correctly functioning pipeline, instructions appear to execute atomically and independently. Inserting NOPs should not change the results produced by a program. This is similar to the concept of flushing a pipeline [2] [4], since by inserting enough NOPs instructions will pass through the pipeline one-at-a-time. This definition allows for the verification of a pipeline implementation without having to know the ISA it executes.

Definition 1: A pipeline implementation is correct if adding NOPs into a program does not change the results produced by the program.

Pipeline hazards arise when an instruction executing in an earlier pipeline stage requires a result from an instruction executing in a later pipeline stage. This is due to either a control dependency or a data dependency between the two instructions. There are three basic techniques for coping with pipeline hazards: bypassing, squashing and stalling [5][6].

The pipeline control logic is comprised of logic to implement bypassing, squashing and stalling. This logic is built from the logic blocks shown in Figure 2. The actions of individual logic blocks are coordinated through a common bypassing/squashing/stalling control input, forming an aggregate control block. The particular function implemented by an aggregate block can be determined from its circuit topology.

All the pipeline control logic is assumed to match one of the topologies in Figure 2.

Property 1: There is no control logic which does not match the any of the topologies in Figure 2.

Pipelines implemented using standard techniques [6] will satisfy Property 1, and a couple of other properties listed below. These properties, which are easily checked, will be used later.

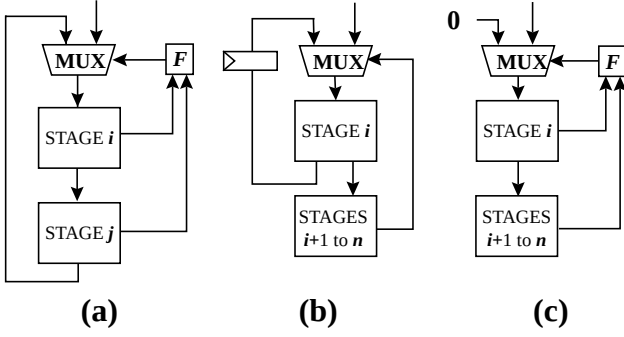


Figure 2. Logic blocks used to build pipeline control

Property 2: A pipeline stage containing a nullified (or squashed) instruction will not activate any pipeline control logic or modify any external-state.

Property 2 states that nullified instructions should not have any effect on other instructions in the pipeline, nor should they change the results produced by previous instructions.

Property 3: If i and j are the deepest stages supporting squash or stall control lines c_i and c_j respectively and $i < j$, then c_j dominates c_i .

Property 3 states that control dependencies are resolved in program order. If both control lines affect the same stage, then either c_i logically cannot be asserted when c_j is, or c_i is a don't care value when c_i is asserted.

Figure 3 shows an example of a pipeline in execution where instruction i_4 squashes the instructions behind it in the pipeline due to a mispredicted taken-branch, and sets the program counter to fetch the target instruction. Pipeline stages two through six have been labeled IF, ID, EX, MEM and WB respectively, corresponding to the five stages of instruction execution presented in [6]. Dividing the pipeline slightly differently than [6], we also have an initial PC stage which contains the next-state equation for the program counter. Notice that although instructions i_5 and i_6 start execution in the PC stage, they do not complete execution in the WB stage.

	Cycle	0	1	2	3	4	5	6	7
S t a g e s	PC	i_5	i_6	i_7	i_8	i_9	i_{10}	i_{11}	i_{12}
	IF	i_4	i_5	X	i_7	i_8	i_9	i_{10}	i_{11}
	ID	i_3	i_4	X	X	i_7	i_8	i_9	i_{10}
	EX	i_2	i_3	i_4	X	X	i_7	i_8	i_9
	MEM	i_1	i_2	i_3	i_4	X	X	i_7	i_8
	WB	i_0	i_1	i_2	i_3	i_4	X	X	i_7

Figure 3. Instruction i_4 is a taken branch

4 Pipeline Templates

Analysis of a pipelined design requires more than identifying and classifying all the control logic into the blocks in Figure 2. Blocks that share the same multiplexer control line act in unison. Unpipelining analysis involves understanding how these blocks work together to control the flow of instructions through the pipeline.

A pipeline interlock, for example, is implemented using one logic block to squash the instruction in stage i and introduce a bubble into the pipeline, while other logic blocks stall all instructions executing in stages above stage i . The blocks all share the same control line that is asserted to trigger the interlock. In the case of an exception, multiple blocks of logic will squash instructions executing in different stages. Again, all the blocks share the same control line that is asserted when an exception occurs; the control line is also used to select the exception program counter.

Both the pipeline deconstruction and the verification steps need to know the function of each control line in the pipeline. Pipeline deconstruction must know how to transform the aggregate logic associated with a control line. Pipeline verification must understand what effect transforming the logic has on the flow of instructions through the pipe.

To match aggregate logic, identified by a shared control line, with a particular function and the knowledge required for pipeline deconstruction and verification, we use a *pipeline template*. A pipeline template is akin to a regular expression for matching logic networks and embodies the design intent of a certain circuit topology. An example of the template for matching the logic to squash instructions on a mispredicted branch as in Figure 3 is shown in Figure 4. Templates are parameterized; for instance, a single template matches the control logic implementing a pipeline interlock at any stage in a pipeline of arbitrary depth.

Each aggregation of control logic in a pipeline must match against a template. Once an aggregation of control logic is associated with a template, the information required for pipeline deconstruction and verification can be requested from the template.

The use of templates makes it easy to extend the verification system. If an exotic pipeline uses an aggregation of control logic not recognized by our system, the required template can be added. Creating such a template should not be difficult for the designers, since it simply reflects an understanding of how they designed the pipeline. However, our system already contains templates that match many of the standard control mechanisms for coping with pipeline hazards. Assuming an appropriate template exists, matching against it is done automatically.

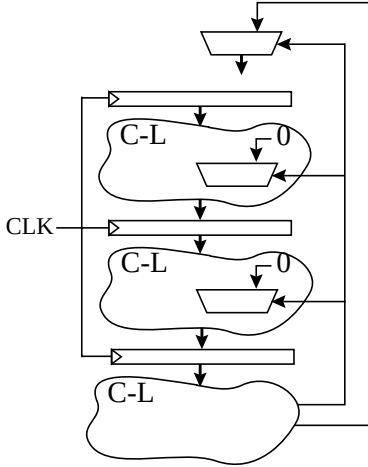


Figure 4. Template for squashing instructions on a mispredicted branch

5 Deconstruction

Pipeline deconstruction transforms a pipeline of depth n to one of depth $n-1$ by merging stages n and $n-1$. If it were not for pipeline hazards, this would be a very simple transformation that just removes the latch between stages n and $n-1$. However, the feedback required by the pipeline control logic to cope with hazards complicates things. In order to merge stages n and $n-1$, any control logic that uses values from stage n to deal with a hazard may need to be modified to ensure that it still deals with the hazard correctly.

Pipeline deconstruction is guided by knowledge of how pipeline control has been implemented; the first step in pipeline deconstruction is to look up the templates that correspond to each aggregation of control logic using a value from stage n . The template for each aggregate indicates how the control inputs should be modified. Control logic can be either left unchanged, removed, or adjusted (replace control input) as shown in Figure 5.

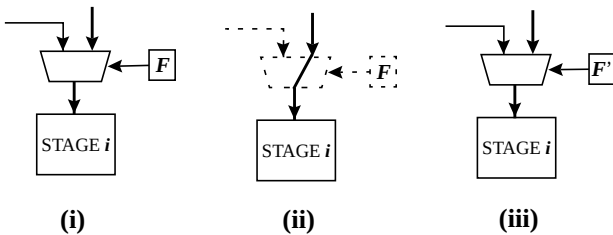


Figure 5. (i) unchanged, (ii) removed, (iii) adjusted

Once the control logic has been modified, the latch between stages n and $n-1$ can be removed. For next-state equations, this corresponds to replacing all instances of current-state variables in the next-state equations for stage

n with their next-state version. In the next-state equations for stage $n-1$, any next-state variables representing external-state are replaced with current-state versions to avoid creating unlocked feedback.

Figure 6 and Figure 7 shows the six-stage pipeline introduced in Section 3 after two and three iterations of unpipelining. Figure 6 merges the EX/MEM/WB stage of Figure 7 with the ID stage. Notice that the squash happens a cycle earlier and affects one less instruction. A pipeline hazard has been removed.

Cycle	0	1	2	3	4	5	6	7
Stages	PC	i_5	i_6	i_7	i_8	i_9	i_{10}	i_{11}
	IF	i_4	i_5	X	i_7	i_8	i_9	i_{10}
	ID	i_3	i_4	X	X	i_7	i_8	i_9
	EX/ MEM/ WB	i_2	i_3	i_4	X	X	i_7	i_8

Figure 6. DLX pipeline after two iterations of deconstruction.

Cycle	0	1	2	3	4	5	6	7
Stages	PC	i_5	i_7	i_8	i_9	i_{10}	i_{11}	i_{13}
	IF	i_4	X	i_7	i_8	i_9	i_{10}	i_{10}
	ID/ EX/ MEM/ WB	i_3	i_4	X	i_7	i_8	i_9	i_{11}
		i_3	i_4	X	i_7	i_8	i_9	i_{11}

Figure 7. DLX pipeline after three iterations of deconstruction.

Conceptually, pipeline deconstruction removes control logic that deals with pipeline hazards present only in the n state pipeline, and leaves intact control logic that deal with hazards common to both pipelines.

Pipeline deconstruction is a mechanical process. For a correctly implemented pipeline, each iteration of pipeline deconstruction will produce an equivalent pipeline that is one stage shorter. Pipeline deconstruction is repeated until either a bug is found in the pipeline, or until all the pipeline control logic has been eliminated and the pipeline contains only two stages: one stage containing the next-stage equation for the program counter, and one stage containing all the other next-stage equations.

6 Verification

A two-stage pipeline (one of the stages being the PC stage) trivially meets Definition 1 for a correctly implemented pipeline. Since such a pipeline executes only one instruction at a time, inserting NOPs into a program will not change the results produced. NOPs do not modify any program state, and thus there exists no mechanism for

them to change the results produced by any other instructions.

Definition 2: Two pipelines are equivalent if they produce the same results for every program.

The verification task is to prove that an initial n -stage pipeline is equivalent to the two-stage variant produced by pipeline deconstruction. We break this down into $n-2$ sub-tasks; after each application of a pipeline deconstruction transformation we prove that the transformed pipeline is still equivalent. If there is a bug in the pipeline implementation, some iteration of pipeline deconstruction will not produce an equivalent pipeline. Otherwise, if each iteration of pipeline deconstruction produces an equivalent pipeline, then the pipeline is bug free.

Let pipeline P' be the result of applying a single pipeline deconstruction transformation to pipeline P . To prove that P' is equivalent to P we use an inductive argument on the number of execution cycles. We assume that all programs that run for i cycles on P' where $i < k$ produce the same results when run on pipeline P , and then prove that if this is the case, then all programs that run for k cycles on P' must also produce the same results on P .

The results produced by a program are contained in the external-state variables. $v@i$ denotes the value of variable v after i cycles of execution; primed variables belong to P' and unprimed variables belong to P . For each external-state variable in P' , we form an induction hypothesis of the form $v'@k = v@k$ that relates values in P' to those in P . The base cases are true by construction of P' . To prove the inductive cases, we use the next-state equations for P' to expand $v'@k$ so that it uses only values from cycles $i < k$. The induction hypothesis are then substituted in, eliminating all the primed variables. With $v'@k$ expressed only in terms of variables in P , we prove that $v'@k$ and $v@k$ are equivalent. The proof is performed by an automatic decision procedure.

There are three main questions to address in the proof. First, which variables require induction hypotheses? The next-state equations for external-state variables refer to pipeline-state, and these equations are essential to the proof. In order to use them we also need induction hypotheses for some pipeline-state variables.

Second, how are the induction hypothesis formed? As seen from the DLX example in the previous section, it is not obvious. The execution of instructions on pipelines P and P' do not match cycle for cycle.

Third, how are the induction hypotheses proven? They must be expanded and bundled with all the necessary information to create formulas suitable for the automatic decision procedure. We address each of these questions in the following sections.

6.1 Selecting Induction Hypotheses

The next-state equations express a value in cycle i in terms of values in cycle $i-1$. Using the next-state equations, $v'@k$ can be replaced with an expression containing only values in cycles $i < k$. For example, if the next-state equation for v' is: $v'_n = f(w'_c, \dots, z'_c)$, where f is some combination logic function, then, $f(w'@k-1, \dots, z'@k-1)$ can be substituted in place of $v'@k$. Applying the induction hypotheses, unprimed variables can be substituted for primed variables, to derive an expression for $v'@k$ that is entirely in terms of variables from pipeline P .

In addition to induction hypothesis for all the external-state variables, we also need induction hypothesis for some pipeline-state variables. While the next-state equations can be used to substitute pipeline-state variables, feedback in the pipeline guarantees that some pipeline-state variables will always be present no matter how many substitutions are performed. For the substitution process to terminate, certain additional key induction hypothesis are required. Determining these hypotheses is equivalent to determining where to break the feedback paths in the pipeline.

For each of the logic blocks in Figure 2, we break the feedback where it connects to the multiplexor as shown in Figure 8. At each point where the feedback is broken, a new variable and its associated next-state equation is added to the set of next-state equations. The new variable is included in the list of variables for which an induction hypothesis is required. This process breaks all the feedback in the model.

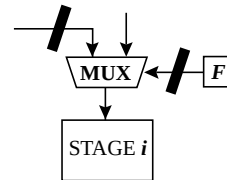


Figure 8. Breaking feedback in pipeline control

However, to ensure that valid induction hypotheses are generated for the new variables, preprocessing of the initial pipeline description is required. The transformations occurs only once, before any pipeline deconstruction transformations are applied. Preprocessing is necessary because in cycles where the output of the control block's multiplexer is a don't-care value, the pipeline will operate correctly even if different feedback values are present in P and P' . Although there is nothing wrong with the original pipeline, induction hypothesis generated from the description would not be satisfied. For example, if the instruction in the pipeline state is a NOP, then the control logic can do

whatever it wants since the value it selects will not be used.

We transform the logic so that the output of the multiplexer is selected only if we care about its value. If we don't care about its value, we bypass the control logic. Figure 8 shows the modifications. With these modifications, we will automatically generate the correct induction hypothesis for the variables introduced by cuts in the feedback. Currently, this transformation is manually performed; although, it would not be hard to automate.

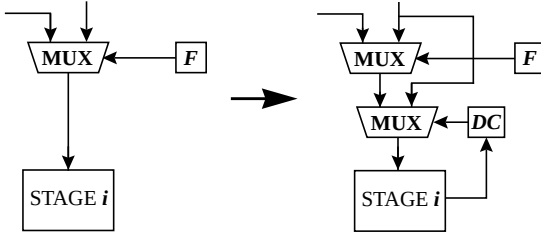


Figure 9. Preprocessing Pipeline Control

6.2 Formulating Induction Hypotheses

The shorter pipeline P' is guaranteed to execute programs in fewer cycles than P . Results produced in stage n of P will be at least a cycle latter than those produced by P' . And, since P' does not have to deal with some of the pipeline hazards in P , the execution rate for P' may be higher. For instance, P' may disrupt the flow of instructions less by resolving branches sooner or by eliminating pipeline interlocks.

First, consider the case where P' has the *same* execution rate as P . Pipeline deconstruction will have only removed latches between stages n and $n-1$, and perhaps some bypassing logic. The effect of this is to cause any variables updated in stage n of P to be updated a cycle earlier in P' . Let $T(v)$ return the stage containing the next-state equation for v . (EQ 2) expresses this variables-in-stage- n -are-one-cycle-early relationship

$$v'@i = \begin{cases} v@i & ; T(v) < n \\ v@(i+1) & ; T(v) \equiv n \end{cases} \quad (\text{EQ 2})$$

Unfortunately, these hypotheses are too simple to apply in the case where the aggregate control logic for stalling or squashing has been changed by an iteration of the unpipelining transformation. Such modifications reflect a pipeline hazard that exists in P , but does not exist in P' . For every cycle pipeline P loses stalling or squashing because of this hazard, P' will end up another cycle further ahead in execution. The exact number of cycles P falls behind P' cannot be predicted in general; it depends on the particular program being executed.

We introduce induction variables to compensate for the difference in execution rates between P and P' . An induction variable s_i^c is associated with each aggregate control block that has been modified by the current iteration of pipeline deconstruction, where c is the control line that defines the aggregate. Each induction variable counts up how many cycles pipelines P and P' are out of synchronization due to differences in the aggregate control blocks defined by their respective c and c' . The two pipelines are re-synchronized by summing up the induction variables that define all the aggregate control blocks modified by pipeline deconstruction.

Let C be the set of all control signals defining such aggregate control blocks. The induction variables are defined as follows. All s_0^c where $c \in C$ are initialized to zero (EQ 3). Whenever an aggregate control block disrupts the pipeline flow, allowing P' to get one cycle ahead of P , the counter for that block is incremented (EQ 4). Otherwise, the counter remains the same (EQ 5).

$$s_0^c \equiv 0 \quad (\text{EQ 3})$$

$$c@ \left(i+1 + \sum_{e \in C} s_i^e \right) \Rightarrow \left(s_{i+1}^c \equiv s_i^c + 1 \right) \quad (\text{EQ 4})$$

$$\neg c@ \left(i+1 + \sum_{e \in C} s_i^e \right) \Rightarrow \left(s_{i+1}^c \equiv s_i^c \right) \quad (\text{EQ 5})$$

For a concrete example of how the induction variables are used to synchronize P and P' , consider the 4-stage pipeline, P , of Figure 6 and the 3-stage pipeline, P' , of Figure 7. For all instruction/stage pairs in P' , a corresponding instruction/stage pair can be found in P . Although after enough cycles, all pairs in P' eventually end up a cycle ahead of their match in P , the advancement does not occur for all stages simultaneously. The PC has already gained a cycle in cycle one, whereas values in the EX/MEM/WB stage don't gain an extra cycle until cycle three. From (EQ 4), if $s_1 = 0$, then $s_2 = 1$. An offset is used to match the change in the value of the induction variable s_i to the advancement of the values in a stage of P' versus P . The PC is offset +1, meaning that $PC'@i = PC@i + s_{i+1}$.

The offset for each stage comes from the template for the aggregate control logic associated with the induction variable. Let $\lambda(c, k)$ return the offset for a control signal c . The inductive hypotheses are:

$$v'@i = \begin{cases} v@ \left(i + \sum_{e \in C} s_i^e + \lambda(e, T(v)) \right) ; T(v) < n \\ v@ \left(i+1 + \sum_{e \in C} s_i^e + \lambda(e, n) \right) ; T(v) \equiv n \end{cases} \quad (\text{EQ 6})$$

For the template in Figure 4, the offsets are shown below:

$T(v)$	1	2	3	4
$\lambda(c, T(v))$	1	1	0	0

Figure 10. Offsets for template in Figure 4

The induction hypotheses relate variables in every cycle of P' to variables in P .

6.3 Proving Induction Hypothesis

The final step is to prove that all the induction hypotheses hold. An automatic decision procedure for a subset of first-order logic expressions [3] automates the proof process. The decision procedure checks if expressions are valid. (An expression p is valid if $\neg p$ is *not* satisfiable.) Thus, an induction hypothesis $v'@k = v@k$ must be combined with the information contained in the next-state equations for P and the invariants which define the induction variables to create a first-order logic formula that is suitable for the decision procedure.

The left-hand-side expression $v'@k$, as described above, is converted into an expression containing only values defined in cycles $i < k$, and the induction hypotheses are applied to replace the primed variables with unprimed variables. Paralleling the expansion of $v'@k$ by substitution of next-state equations for P' , is the expansion of $v@k$ by next-state equations for P . This results in $v'@k = v@k$ being converted to an expression where the left-hand-side and the right-hand-side are pretty similar.

The expression, however, may not be valid as is, since the next-state equations for P' may not be exactly the same as those for P , depending on the control logic removed by pipeline deconstruction. In order to deduce that the left and right-hand sides are equivalent, additional invariants are required. These invariants include (EQ 4) and (EQ 5) for specific values of i , the next-state equations for certain variables in P at specific values of i , and information about the consequences of asserting control signals $c \in C$.

As the pipeline is deconstructed, the differences between P and P' , and the extra invariants required to prove each induction hypothesis are recorded. The removal of bypassing logic requires the inclusion of the next-state equation for the 0-input to the multiplexer. The removal of stalling logic requires the inclusion of the next-state equation for the 1-input to the multiplexer.

Invariants concerning the consequences of asserting control signals allow for the resolution of different synchronization variables that appear on the left and right-hand sides. The necessary information is contained in the template for the aggregate control logic, and the appropri-

ate assertions are automatically constructed for each aggregate control block affected by pipeline deconstruction.

For example, say a control signal br indicating a mispredicted branch is asserted in cycle i and this results in the squashing of instructions in stages $n-1$ to 2. Then for the next $n-2$ cycles, a squashed instruction will occupy stage n . Since br is produced in stage n , br will not be asserted during those cycles. In fact, no control signal will be asserted from any stage that is occupied by a squashed instruction. (EQ 7) shows the invariant generated for the br signal.

$$br@ \left(i + 1 + \sum_{e \in C} s_i^e \right) \Rightarrow \\ \neg br@ \left(i + 2 + \sum_{e \in C} s_i^e \right) \wedge \dots \wedge \neg br@ \left(i + n - 1 + \sum_{e \in C} s_i^e \right) \quad (\text{EQ 7})$$

These invariants are all conjoined. A formula in which the induction hypothesis is implied by the conjunction is built and fed through the automatic decision procedure. If the formulas (proof requirements) for all the induction hypothesis are valid, then the pipelines are equivalent.

7 Experimental Results

The deepest pipeline we have experimented with is a 12-stage pipeline that implements a RISC-style ISA with six operation classes: ALU immediates, 3-register ALUs, branches, jumps, loads and stores. The 12-stage pipeline uses a predict-not-taken strategy to speculatively execute instructions. The branch penalty is five cycles and jumps have a penalty of three cycles. The pipeline has a load-interlock between the load/store stages and the execute stages.

With the exception of the preprocessing as discussed in Section 6.1 the proof was automatic. Pipeline deconstruction and generation of the proof requirements took 8.79s on an SGI Indy with a 200MHz R4400. As is expected, the proof was the most time intensive step. Figure 11 shows the verification time required per iteration of pipeline deconstruction, and the number of induction hypothesis generated. As the pipeline is deepened, the number of induction hypothesis increases. However, the time required to verify each iteration of pipeline deconstruction is not related to the depth of the pipeline; it is related to the complexity of the control logic removed in a given iteration.

The proofs were performed by SVC [3], an automatic tool developed at Stanford University for deciding the validity of expressions in a subset of first-order logic. The decision procedure in SVC was augmented with heuristics

to cache successful case splits (case splits defined in [3]). This helped SVC find more optimal orderings for case splits, significantly improving its performance and extending the complexity of the pipelines we could handle.

Stages Merged	Execution Time	# Ind. Hypotheses
11 & 12	46.9s	26
10 & 11	11.4s	22
9 & 10	4.8s	21
8 & 9	3.9s	20
7 & 8	6.0s	19
6 & 7	6.1s	7
5 & 6	4.4s	7
4 & 5	2.4s	6
3 & 4	1.5s	6
2 & 3	1.6s	6

Figure 11. Execution time to verify 12-stage pipeline

8 Limitations

Although more automatic and scalable than other methods, unpipelining is less general in its applicability. It is not suitable for verifying pipelines that implement instruction set architectures (ISAs) which expose pipeline artifacts such as delayed loads and branches; though, for such cases it can still be helpful when applied in conjunction with other methods. While techniques such as [1][2][4] are most effective on pipelines that *do* expose pipeline artifacts and are much less effective on pipelines that do not, unpipelining is best suited for pipelines that *do not* expose pipeline artifacts.

Unpipelining relies on the use of a standard design style in order to deconstruct a pipeline. Other methods place no restrictions on the design style; however, they require designers to manually construct certain non-trivial specifications and mapping functions [1][2][4]. There is a clear trade-off between design freedom on one hand and ease of verification on the other. The ability of unpipelining to understand how a pipeline has been implemented is key to its superior scalability.

Currently, the don't care conditions for bypass control logic must be determined manually. This step could be automated. And, as with other proposed methods, unpipelining requires the manual abstraction of datapath and other unimportant (for the purpose of verifying a pipeline implementation) components.

9 Conclusions

We have described an automatic technique based on unpipelining for verifying microprocessor pipelines. This technique extends unpipelining by making it more general

and more automatic. We have implemented our technique using an automatic decision procedure and demonstrated that the technique can be used to prove the correctness of deep, complex pipelines.

The primary advantages of our technique are that it requires no specification, that it is scalable to deeper pipelines, and that it handles the pipeline control logic for squashing and stalling instructions naturally. We have demonstrated that our technique can verify a class of pipelines that are deeper and more complex than can be verified using other methods. This technique gains its advantage from understanding pipeline implementation techniques, and is thus more restrictive than other methods in that certain standard design practices need to be adhered to. Although it is not directly applicable to pipelines that implement ISA's with artifacts such as delayed loads or delayed branches, these artifacts have been removed in the most modern RISC ISA's such as the DEC Alpha AXP architecture.

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] C. Barrett, D. L. Dill and J. Levitt, "Validity Checking for Combinations of Theories with Equality" in *Proceedings of The International Conference on Formal Methods in Computer-Aided Design*, Palo Alto, CA, November 1996.
- [4] J. R. Burch, "Techniques for Verifying Superscalar Microprocessors" in *Proceedings of The 33rd ACM/IEEE Design Automation Conference*, Las Vegas NV, June 1996.
- [5] J. Levitt and K. Olukotun, "A Scalable Formal Verification Methodology for Pipelined Microprocessors" in *Proceedings of The 33rd ACM/IEEE Design Automation Conference*, Las Vegas NV, June 1996.
- [6] J. Hennessy and D. Paterson, *Computer Architecture: A Quantitative Approach*, 2nd ed. Morgan Kaufman, 1996.
- [7] 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.
- [8] 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.
- [9] K. McMillan, "Fitting Formal Methods into the Design Cycle," in *Proceedings of 31st Design Automation Conference*, San Diego CA, June, 1994.
- [10] M. Srivas and M. Bickford, "Formal Verification of a Pipelined Microprocessor" *IEEE Software*, vol. 7, no. 5, pp. 52–64, Sept. 1990