Reasoning About Pipelines with Structural Hazards

Mark Aagaard* and Miriam Leeser

School of Electrical Engineering
Cornell University
Ithaca, NY 14853
USA

Abstract. We have developed a formal definition of correctness for pipelines that ensures that transactions terminate and satisfy a functional specification. This definition separates the correctness criteria associated with the pipelining aspects of a design from the functional relationship between input and output transactions. Using this definition, we developed and formally verified a technique that divides the verification of a pipeline into two separate tasks: proving that the pipelining circuitry meets the pipelining correctness criteria and that the datapath and control circuitry meet the functional specification. The first proof is data independent (except for pipelines that use data-dependent control). The second proof is purely combinational: there is no notion of time and each possible input transaction can be dealt with independently. In addition, we have created a framework that structures and simplifies the proof of the pipelining circuitry.

1 Introduction

The work presented here is part of a larger effort to develop systematic techniques for the specification, design, and verification of large-scale, complex pipelined circuits [AL93]. We have concentrated on incorporating design features that are used in state-of-the-art high-performance microprocessors: super-scalar instruction issue, out-of-order completion and data-dependent control. These features are found in microprocessors such as the DEC Alpha [McL93], HP PA-Risc [AAD+93], IBM RS/6000 [Mis90], and Intel Pentium [AA93] and lead to implementations that contain structural hazards.

Structural hazards increase the difficulty of design and verification, because errors related to non-termination and interference between transactions may occur. They also complicate the specification of a pipeline, because transactions may have variable latencies (the transit time through a pipeline) and may exit in a different order than they entered. In pipelines without structural hazards, verifying termination and freedom from interference is trivial. Thus, the bulk

* Now at Department of Computer Science, University of British Columbia, Vancouver, BC, Canada

of the verification effort is concerned with verifying a datapath in which computations occur over some interval of time. When confronting pipelines with structural hazards, even specifying the intended behavior and verifying that all transactions will terminate can be a significant challenge.

1.1 Our Work

We often reason about hierarchical pipelines, where the datapaths for the stages may themselves be pipelines. We say that pipelines are composed of *segments* and that *stages* are segments that can contain at most one transaction (*i.e.* they are at the lowest level of the structural hierarchy).

Modern pipelined microprocessors contain: an instruction pipeline that fetches, decodes, and issues instructions; and several execution pipelines (*e.g.* integer, floating-point, and load/store). All of the floating-point pipelines, and most of the instruction and integer pipelines that we have found, contain structural hazards. In addition to structural hazards, instruction pipelines have data and control hazards. If a pipeline is free from *structural conflicts*[2] then *individual transactions* are able to proceed through the pipeline correctly. Control and data hazards cause problems with *dependencies between transactions*. Our belief is that before we can discuss dependencies between transactions, we should be able to reason effectively about individual transactions. The work presented here supports reasoning about structural hazards. Our preliminary efforts to extend this work to data and control hazards appears promising: we believe that the extensions can be added by building on our work with structural hazards.

We separate the general idea of correctness into two properties: *pipelining*, every transaction that wishes to enter the pipeline is eventually able to do so and there is a one-to-one mapping between transactions that enter and exit the pipeline; and *functionality*, every output transaction has the correct value. This definition of correctness allows us to separate the verification of a pipeline into a pipelining and a functionality proof. The pipelining proof is data-independent for almost all pipelines and the functionality proof is purely combinational.

The pipelining correctness criteria are the same for every pipeline. Because of this, even for pipelines which appear to be quite different, there is a great deal of similarity in how we verify that there are no structural conflicts. We have organized these similarities into a framework that captures reasoning common to these proofs. Using the framework allows the bulk of the verification of an individual pipeline to concentrate on the datapath.

The framework defines four parameters that are instantiated for individual pipelines: *Protocol* schemes describe how transactions are transferred between stages in the pipeline; *Arbitration* schemes specify how to handle collisions in the pipeline; *Control* schemes determine how transactions are routed through the pipeline and how stages know what operation to perform; and *Ordering* schemes

[2] Following the terminology of Tahar [TK94], which is based on that of Milutinovic [Mil89], we say that a pipeline is free from structural conflicts if it handles its structural hazards correctly or has no structural hazards.

describe a method for matching up a transaction as it leaves the pipeline with the corresponding input transaction.

We have derived specifications for the parameters and proved that any pipeline that meets the specifications is guaranteed to meet our pipelining correctness criteria. This proof was done on paper using a strongly-typed higher-order logic in a style that is compatible with interactive tactic-oriented proof development systems such as Nuprl, HOL, PVS, and Coq. Our syntax is based most closely on that used in Nuprl and SML. Using the framework divides the proof that a pipeline is free from structural conflicts into separate and largely independent tasks. The specifications for the parameters are very general, so as to allow for innovative design solutions and evolving technology.

In order to reason about pipelines in general and then apply the results to specific pipelines, we developed a model of a pipeline that is composed of generic segments. We defined a set of virtual functional units and signals in a generic segment and then derived behavioral specifications for these units. Using the framework to verify a pipeline is done by defining the correspondence between the hardware in the real pipeline and the virtual functional units and then proving that the circuits in the real pipeline meet the specifications for the corresponding virtual functional units.

We say the functional units and signals are "virtual," because a given pipeline may not implement all of the units in hardware. For example, a uniform-unifunctional pipeline (all transactions follow the same path and each segment can perform only one operation) does not need any control hardware. The verification of such a pipeline would define the virtual functional units for control to be constants that produce outputs of type *unit*, a type that has only one member.

1.2 Related Work

Previous research on verifying pipelined microprocessors using proof-development systems includes Srivas's verification of the Mini-Cayuga using Clio [SB90] and Saxe's verification of a very small microprocessor using the Larch Proof Assistant [SGGH92]. Saxe's example has also been done by Alex Bronstein using the Boyer-Moore theorem prover Nqthm [BT90]. Lessons learned from these efforts include Srivas's and Bronstein's use of equivalence mappings between the behavior of pipelined and unpipelined microprocessors.

Windley has developed a methodology for verifying microprocessors based on generic interpreters [Win91] and is extending this methodology to handle pipelined designs [WC94]. This methodology decomposes verifying a pipeline into a series of abstraction mappings between the gate level and the programmer's model. Incorporating pipelining into generic interpreters provides a systematic approach for specifying and verifying pipelined microprocessors, but complicates the abstraction mappings.

Tahar is using HOL to verify Hennessy and Patterson's academic microprocessor, the DLX [TK93]. As in the work presented in this paper, Tahar also separates the verification effort into pipelining and functionality concerns, but

we have a stronger definition of correctness and have proved that the pipelining and functionality concerns can be separated.

Model checking systems have also been used to verify pipelines. Beatty has developed the theory of "marked strings", which provides a basis for the automated verification of circuits, including pipelines [Bea93]. Seger has used the Voss system to verify an integer pipeline that is reflective of those in a modern RISC microprocessor [Seg93]. The limiting factor in using model-checking techniques to verify a pipeline is the size of the state-space. To mitigate this, Burch [BD94] has separated the pipelining concerns from the functionality concerns. His work provides a highly automated approach for verifying pipelining circuitry, but he does not reason about composing pipelining and functionality correctness results.

A number of the efforts listed here have dealt with control and/or data hazards, but none of the pipelines have significant structural hazards. The only other formal work we are aware of that deals with realistic structural hazards is Harcourt's use of SCCS to formally specify microprocessors and derive instruction schedulers [HMC94]. This work is complementary to ours, in that Harcourt works upward from formal specifications of microprocessors to schedulers and we verify implementations against specifications.

1.3 Outline of Paper

In Section 2 we discuss our formal definition of correctness for pipelines. Section 3 describes how we use the framework to verify pipelines. Sections 4 and 5 include an introduction to the four parameters to the framework, their specifications, and a set of commonly used instantiations of the parameters. Section 6 contains an overview of how the framework can be used to characterize and verify the floating-point pipeline of the Adirondack, a fictitious super-scalar RISC microprocessor.

2 Correctness

We have defined what it means for a pipeline to be free from structural conflicts and to meet a functional specification. This definition ensures that every transaction that wishes to enter the pipeline is able to do so, every transaction that enters the pipeline eventually exits, and every transaction that exits the pipeline has the correct data. This definition of correctness is captures in the predicate *satisfies*, shown in Definition 1 and displayed using the symbol $\sqsupset$.

Definition 1: *Pipeline satisfies a specification*

$$
\begin{aligned}
&(\mathit{Pipe},\ \mathit{Match},\ \mathit{Constraints}) \sqsupset \mathit{Spec} \mathrel{\hat{=}} \\
&\quad \forall I,\ O\ .\ (\mathit{Pipe}\ I\ O)\ \&\ (\mathit{Constraints}\ I\ O)\ \&\ (\mathit{envOk}\ I\ O) \implies \\
&\qquad \mathit{PipeliningOk}\ \mathit{Match}\ I\ O\ \& \\
&\qquad \forall t_i,\ t_o\ .\ \mathit{Match}\ I\ O\ t_i\ t_o \implies \mathit{Spec}\ (I\ t_i)\ (O\ t_o)
\end{aligned}
$$

We parameterize *satisfies* by the implementation of a pipeline (*Pipe*), a relation for matching corresponding input and output transactions (*Match*), a possible set of constraints on the environment (*Constraints*), and a functional specification (*Spec*). The first part of the definition uses *PipeliningOk* (Definition 2) to ensure that the pipelining aspects of correctness are met. This definition is the same for every pipeline, which, as shown in Section 3.2, allows us to greatly simplify the pipelining part of the verification. The second part ensures that the functionality aspects of correctness are met. The correctness criteria for functionality says that when *Match* finds a matching input and output time (t_i and t_o), then the input transaction at t_i and the output transaction at t_o must satisfy the functional specification (*Spec*).

We parameterize *satisfies* by *Match*, because the temporal relationship between input and output transactions varies from pipeline to pipeline. Some pipelines work correctly only if their environment obeys some constraints, such as being always able to receive output transactions. We support this by parameterizing *satisfies* with *Constraints*. A functional specification (*Spec*) is a relation over an input transaction and an output transaction. It defines the computation that the pipeline is meant to perform. It is purely combinational and is not concerned with pipelining aspects of correctness.

In the definition of *satisfies*, the pipeline (*Pipe*) interacts with an environment through an input stream (*I*) and an output stream (*O*). The predicate *envOk* ensures that environment conforms to several requirements, such as every transaction that wants to exit the pipeline is eventually able to do so.

A pipeline is free from structural conflicts if: every transaction that wants to enter the pipeline is eventually able to do so, every transaction that enters eventually exits, and every transaction that exits the pipeline has entered. These criteria are captured in *PipeliningOk* (Definition 2). The relation *canEnter* guarantees that transactions that wish to enter the pipeline are eventually able to do so. In the second clause of *PipeliningOk* we use the matching relation for the pipeline (*Match*) to simplify the second and third correctness criteria to: *Match* defines a one-to-one mapping between input and output transactions (*isOneToOne*. These properties guarantee that the pipeline is free from deadlock and livelock and that transactions are not created inside the pipeline.

Definition 2: *Pipelining correctness criteria*

PipeliningOk Match I O $\hat{=}$

(*canEnter I O*) & (*isOneToOne Match I O*)

3 Verification of Pipelines

In this section we introduce our techniques for formally verifying pipelines using *satisfies*. The process is the same for all pipelines, so we illustrate it with a

canonical pipeline (*Pipe*), matching relation (*Match*), constraints (*Constraints*), and specification (*Spec*). We begin with the proof goal that the pipeline, matching relation, and constraints satisfy the specification (Equation 1).

$$(Pipe,\ Match,\ Constraints) \sqsupset Spec \tag{1}$$

In a naive proof of Equation 1, we would unfold *satisfies* to produce two goals: one to show that the pipeline is free of structural conflicts and one to show that the pipeline and matching relation imply the functional specification (Figure 1). These two goals separate the *pipelining* and *functionality* aspects of the verification, but the implementation of the pipeline (*Pipe*) appears in the functionality goal. This means that the functionality proof must deal with pipelining concerns, such as potential collisions and out-of-order termination.

$\vdash (Pipe,\ Match,\ Constraints) \sqsupset Spec$
BY `Unfold` *satisfies* (* Definition 1 *)

- $(Pipe\ I\ O)\ \&\ (Constraints\ I\ O)\ \&\ (envOk\ I\ O)$
  $\vdash PipeliningOk\ Match\ I\ O$

- $(Pipe\ I\ O)\ \&\ (Constraints\ I\ O)\ \&\ (envOk\ I\ O)$
  $\vdash Match\ I\ O\ t_i\ t_o \implies Spec\ (I\ t_i)\ (O\ t_o)$

Fig. 1. Direct verification of canonical example; unfold *satisfies*

Rather than following the naive approach just described, we have proved two theorems that completely separate the pipelining and functionality proofs and then greatly simplify the pipelining proof. In Section 3.1 we introduce the *combinational representation*, which leaves us with a functionality proof that is purely combinational and a pipelining proof that is almost always data-independent. In Section 3.2 we show how to simplify the pipelining proof using the specifications for the four parameters to the framework (protocol, arbitration, control, and ordering).

3.1 Using A Combinational Representation To Verify A Pipeline

To remove all aspects of pipelining from the functionality proof we use Theorem 1 to introduce a *combinational-logic representation* (*Comb*) of the pipeline. A combinational representation captures the input/output functionality but not the pipelining aspects of the implementation. This is because it reflects the datapath and control circuitry of the implementation, but not the pipelining circuitry. As an abbreviation, we package the implementation of our example pipeline (*Pipe*), matching relation (*Match*), and constraints (*Constraints*) as *PipeRecord*

When using Theorem 1 to verify a pipeline, we introduce the combinational representation and have the two goals shown in Figure 2, rather than those shown in Figure 1. The first goal, which we refer to as the pipelining goal, is to

Theorem 1: *Transitivity of satisfies*

$$\vdash \forall \mathit{PipeRecord},\ \mathit{Comb},\ \mathit{Spec}\ .\ \mathit{PipeRecord} \sqsupset \mathit{Comb}\ \&\ (\forall T_i,\ T_o\ .\ \mathit{Comb}\ T_i\ T_o \implies \mathit{Spec}\ T_i\ T_o) \implies \mathit{PipeRecord} \sqsupset \mathit{Spec}$$

prove that the pipeline *satisfies* the combinational representation. This proof is much easier than showing that the pipeline *satisfies* the high-level specification (*Spec*), as in Figure 1. The combinational representation is closely related to the pipeline and the proof is data-independent (except for pipelines that use data-dependent control). In contrast, specifications are generally unrelated to the structure of the pipeline and reasoning about them is highly data-dependent. The second goal in Figure 2, which we refer to as the functionality goal, is to show that the combinational representation implies the specification.[3] This goal is purely combinational and can be solved using standard hardware verification techniques.

$\vdash \mathit{PipeRecord} \sqsupset \mathit{Spec}$
BY Theorem 1

$\vdash \mathit{PipeRecord} \sqsupset \mathit{Comb}$ (* pipelining *)

$\vdash \forall T_i,\ T_o\ .\ \mathit{Comb}\ T_i\ T_o \implies \mathit{Spec}\ T_i\ T_o$ (* functionality *)

Fig. 2. Verification of canonical example; introduce combinational representation

3.2 Using the Framework to Verify a Pipeline

In this section we introduce the framework to simplify the pipelining proof (the first goal in Figure 2). We use Theorem 2 to divide the proof that a pipeline *satisfies* its combinational representation into four subgoals (Figure 3): the core (datapath) for each segment is valid (*CoresOk*), the constraints of the segments are met (*ConstraintsOk*), the combinational representation is an accurate representation of the pipeline (*CombOk*), and the pipelining circuitry that glues the segments together meets the specifications of the framework parameters (*FworkParamsOk*).

Because we work with hierarchical pipelines, the core of a segment may itself be a pipeline. The core of each segment has an associated matching relation and combinational representation. The definition of *CoresOk* says that the core of

[3] We use the convention that capital Ts (*e.g.* T_i and T_o) are for transactions and lowercase Ts (*e.g.* t_i and t_o) are for times.

Theorem 2: *Pipelines that meet framework specifications satisfy their combinational representations*

$\vdash \forall$ *Pipe, Match, Constraints, Comb* .
(*CoresOk Pipe*) &
(*ConstraintsOk Pipe Constraints*) &
(*CombOk Pipe Comb*) &
(*FworkParamsOk* (*Pipe, Match, Constraints*)) $\Longrightarrow$
(*Pipe, Match, Constraints*) $\sqsupset$ *Comb*

$\vdash$ (*Pipe, Match, Constraints*) $\sqsupset$ *Comb*
BY Theorem 2

$\vdash$ *CoresOk Pipe*

$\vdash$ *ConstraintsOk Pipe Constraints*

$\vdash$ *CombOk Pipe Comb*

$\vdash$ *FworkParamsOk* (*Pipe, Match, Constraints*)

Fig. 3. Verification of canonical example; use framework to show that pipeline satisfies combinational representation

each segment *satisfies* its combinational representation, the data in an output transaction does not change while waiting for its request to be accepted, and the combinational representation of the core is functional (has equal outputs for equal inputs).

The second goal in Figure 3 is to prove that the constraints for each segment are met. This condition arises because in *satisfies* a pipeline may put constraints on its environment and the core of a segment may be a pipeline. Thus, using a segment in a pipeline requires showing that the pipeline meets the constraints of the segment. In the third goal, the relation *CombOk* relates the combinational representation (*Comb*) to *Pipe*. It requires that *Comb* is equivalent to composing the combinational representations of the segments. For the fourth goal, the framework defines four parameters (protocol, arbitration, control, and ordering) that characterize pipelining circuitry. Each of these parameters has an associated set of specifications and *FworkParamsOk* says that the pipeline meets these specifications.

3.3 Summary of Verifying a Pipeline

When verifying a pipeline we use Theorem 1 to introduce a combinational representation and separate the pipelining and functionality aspects of verification. We then use Theorem 2 to simplify the pipelining proof. This leaves us with the five proof obligations listed in Table 1.

Table 1. Proof obligations when verifying a pipeline

CoresOk PipeRecord
Datapaths and control circuitry of segments are valid
ConstraintsOk PipeRecord
Constraints of segments are met
CombOk Pipe Comb
Combinational representation accurately represents pipeline
FworkParamsOk PipeRecord Comb
Pipelining circuitry meets the specifications of the framework parameters
$\forall\ T_i,\ T_o\ .\ Comb\ T_i\ T_o \implies Spec\ T_i\ T_o$
Combinational representation implies specification

For a stage (a segment that can contain at most one transaction), the pipelining circuitry is so simple that it is almost always trivial to prove that it conforms to *CoresOk*. For a segment, we use a theorem (not shown, because it is almost identical to Theorem 2) that says that if a pipeline meets the four antecedents of Theorem 2, then the pipeline meets all of the requirements in *CoresOk*. This means that the pipeline can be used as the core of a segment in a larger pipeline. Thus, the first goal is solved in the normal progression through the structural hierarchy of a large pipeline (*e.g.* a microprocessor composed of instruction, integer, floating-point, and load/store pipelines).

The second obligation, that the constraints of the segments are met, is usually quite easy to solve. Many segments do not put any constraints on their environment. Every constraint that we have found affects only one segment or the environment, and the constraints are direct consequences of the behavior of the segment or environment.

The third goal in Table 1 is to prove that the combinational representation of the pipeline is equivalent to the composition of the combinational representations of the segments. We systematically build the combinational representation of a pipeline by composing the combinational representations of the segments according to the paths that transactions follow. Thus, just as with the first goal, the third goal is solved simply by following our standard techniques.

The fourth goal is to prove that the specifications of the framework parameters are met. The informal specifications of the parameters are given in Section 4. In addition, we have found a set of instantiations of the parameters that is sufficient to characterize the pipelining circuitry of many microprocessors (Section 5). These instantiations guide the proofs that a pipeline meets the specifications of the parameters.

The fifth goal is for the functionality proof. It requires showing that the combinational representation of the pipeline implies the specification. This goal is purely combinational and can be solved without any pipeline-related reasoning.

4 Specifications for Framework Parameters

The specifications for a protocol scheme ensure that when two segments agree to transfer a transaction, the transaction is transferred correctly. In the arbitration specifications we check that when one segment wants to transfer a transaction to another, the second segment eventually agrees to receive the transaction. By combining the protocol and arbitration specifications, we prove that transactions make progress and flow through the pipeline. The control specifications require that all paths through the pipeline lead to an exit and that transactions follow the same path through the implementation and the combinational representation. From this we know that transactions will traverse the correct path and then exit. The ordering specification says that when a transaction exits, we can match it up with its corresponding input transaction. This allows us to check that corresponding input and output transactions meet the functional specification. There are a total of eighteen specifications, of which only six require significant effort to prove for most pipelines. To give a flavor of the specifications, we describe them textually and show several lemmas that were proved using the specifications and the requirements for cores of segments. At the end of this section, Table 2 summarizes the specifications.

4.1 Protocol

The purpose of a protocol scheme is to move transactions from one segment to the next in a pipeline. In order to transfer a transaction between segments, the segment that is to receive the transaction needs to know that the other segment wants to send it a transaction. Conversely, the sending segment needs to know if the receiving segment is able to receive the transaction. We have named these two properties, the desire to send and the ability to receive, "request" and "accept." We use the protocol specifications to prove Lemma 1, which says that at time t, a transfer must occur from segment s_0 to s_1 if s_0 sends a request to s_1 and s_1 accepts the request.

Lemma 1: *Correctness of transfers between segments*
$\vdash \forall\ s_0,\ s_1,\ t\ .\ (req\ (s_0,\ s_1)\ t)\ \&\ (acc\ (s_0,\ s_1)\ t) \implies xfr\ (s_0,\ s_1)\ t$

4.2 Arbitration

The specifications for an arbitration scheme are concerned with ensuring that every request is eventually accepted, as shown in Lemma 2. The predicate *holdUntil* is used to say that if $req\ (s_0,\ s_1)$ is true at time t_r, then there is exactly one time t_a such that $req\ (s_0,\ s_1)$ is true from t_r to t_a and t_a is the first time after t_r that $acc\ (s_0,\ s_1)$ is true.

Lemma 2: *Every request is eventually accepted*
$\vdash \forall\ s_0,\ s_1,\ t_r\ .\ \exists\ t_a\ .\ holdUntil\ (req\ (s_0,\ s_1))\ t_r\ (acc\ (s_0,\ s_1))\ t_a$

We separate the arbitration specifications into two parts: the highest priority request to a segment is eventually accepted and every request to a segment becomes the highest priority request. These two properties are related to deadlock and livelock respectively. If the highest priority request to each segment is always accepted, then the pipeline can not deadlock. Adding the requirement that every request becomes the highest priority request ensures that every request is eventually accepted, and hence livelock is prevented.

4.3 Control

A control scheme determines how a segment decides what operation to perform on each transaction and where to send the transaction when it is done in the segment. The control specifications allow us to prove that every path through a pipeline leads to an exit and that transactions follow the same path through the implementation and combinational representation of a pipeline. We require that every path through a pipeline leads to an exit, because without this requirement it would be valid for a pipeline to contain paths that loop through the same set of segments an infinite number of times. Such paths would not produce any output transactions, despite the fact that the pipeline is working "correctly."

Lemma 3: *Transactions transfer to the correct next segment*
$\vdash \forall s_0,\ s_1,\ t_0,\ t_1,\ T\ .$
$\quad match\ s_0\ (t_0,\ t_1)\ \&$
$\quad segComb\ s_0\ (transP\ s_0\ t_0)\ (T,\ s_1)\ \Longrightarrow$
$\quad\quad xfr\ (s_0,\ s_1)\ t_1$

Lemma 3 says for each step in a path, a transaction will transfer to the same next segment and its combinational representation. Formally, the lemma says that if a transaction transfers into segment s_0 at time t_0 and the combinational representation of s_0 produces a transaction T and selects s_1 as the next segment, then in the implementation the transaction will transfer from s_0 to s_1. We relate the combinational representation of s_0 to its implementation by using the input transaction to s_0 at t_0 $(transP\ s_0\ t_0)$ as the input transaction to the combinational representation and using the matching relation for s_0 to detect when the transaction exits from s_0.

4.4 Ordering

Ordering schemes are used to verify that output transactions contain the correct results. To do this, we match up an output transaction with the input transaction

that caused it. For example, if a transaction contains an add instruction, we need to check that the data in the output transaction is the sum of the two operands in the input transaction. Each pipeline defines a matching relation (*Match*), which takes two times (t_0) and (t_n) and returns true if the input transaction at t_0 results in the output transaction at t_n.

The relation *isOneToOne*, which is part of the pipelining correctness criteria, says that a matching relation defines a one-to-one mapping between the input and output transactions of a pipeline. The framework uses the ordering specification to prove that a matching relation defines a one-to-one mapping between input and output transactions and to prove that if the matching relation matches an input and output transaction then they satisfy the combinational representation.

Table 2. Summary of major specifications

Protocol	Transactions are transferred between segments correctly
Arbitration	The highest priority request to each segment is eventually accepted
	Every request to a segment becomes the highest priority request
Control	All paths through the pipeline lead to an exit
	The implementations of the control circuitry imply their specifications
Ordering	The matching relation for the pipeline finds corresponding input/output transactions

5 Common Instantiations of Framework Parameters

We have found a number of commonly used instantiations of the framework parameters. These instantiations are sufficient to characterize the instruction, integer, and floating-point pipelines of the DEC Alpha 21064, HP PA-Risc 7100, IBM RS/6000, Intel Pentium, MIPS R4000, Motorola 88110, and PowerPC 601. At the end of this section, Table 3 summarizes the instantiations.

5.1 Protocol

For some pipelines, we can guarantee that every request will be immediately accepted. These pipelines are free of structural hazards and use a *transit* protocol scheme, so called because transactions can "transit" through the pipeline. In the second scheme, called *general* because it matches the general specification of protocol correctness, transactions are allowed to proceed until just before a

collision. When a transaction cannot proceed any further without colliding, it stalls until the potential collision is cleared.

5.2 Arbitration

In a *degenerate* arbitration scheme segments receive input transactions from only one segment. If a segment is connected to multiple input segments, but we can show that only request will be active at a time, then we have an *exclusive* arbitration scheme. The name is derived from the observation that each request is guaranteed to be the exclusive request to the segment. In this arbitration scheme the segment simply selects whichever request is active.

For pipelines that allow multiple simultaneous requests to the same segment, we need to provide a method for prioritizing the requests. In all pipelines that we have found, we can assign *static* priorities to requests based upon the segment that the request comes from.

5.3 Control

If all transactions use each segment in a pipeline at most once, and every segment sends transactions to another segment or to an exit port, then we know that all paths are finite and reach an exit. We refer to this as a *no-loops* control scheme. The only control schemes that are non-trivial to verify are those in which transactions may use segments multiple times. We need to prove that none of these loops can be repeated an infinite number of times. In all of the pipelines that we surveyed, there is fixed upper bound on the number of iterations that transactions can pass through each loop. So, in practice, control schemes are straightforward to verify.

5.4 Ordering

The ordering schemes described here are listed in order of increasing generality. In most cases, it is easiest to use the most specific ordering scheme that is applicable to a pipeline. The simplest way to match input and output transactions is if the pipeline has a *uniform latency*. In these pipelines all transactions will exit the pipeline a given number of cycles after they enter the pipeline. The next simplest case is pipelines where transactions exit in the same order that they enter the pipeline. This is an *in-order* scheme.

In all pipelines that we have found where transactions may exit out of order, we can assign tags to transactions in such a way that there is only one transaction with a given tag in the pipeline at a time (*tags-unique*) or transactions with the same tag exit in-order (*tags-in-order*). In the pipelines that we surveyed, either the opcode of the transaction or the destination of the result of the operation (*e.g.* a register or memory address) is used as a tag.

Table 3. Instantiations of framework parameters for commercial microprocessors

		Instruction							Integer							Floating Point						
		DECchip 21064	HP PA-Risc 7100	IBM RS/6000	Intel Pentium	MIPS R4000	M88110	PowerPC 601	DECchip 21064	HP PA-Risc 7100	IBM RS/6000	Intel Pentium	MIPS R4000	M88110	PowerPC 601	DECchip 21064	HP PA-Risc 7100	IBM RS/6000	Intel Pentium	MIPS R4000	M88110	PowerPC 601
Proto	Transit								●	●	●		●	●	●	●	●	●	●	●	●	●
	General	●	●	●	●	●	●	●				●										
Arb	Degenerate		●			●	●			●		●		●					●			
	Exclusive			●				●	●		●		●		●	●	●	●		●	●	●
	Static	●			●																	
Ctrl	No loops	●	●	●	●	●	●	●		●		●		●					●			
	Loops								●		●		●		●	●	●	●		●	●	●
Ordering	In order		●			●				●		●		●	●			●	●			●
	Tags: Unique								●		●		●			●	●			●	●	
	Tags: In order	●		●	●		●	●														

6 Floating Point Pipeline

This section describes the floating-point pipeline from a fictitious RISC superscalar microprocessor, the Adirondack (ADK). The ADK is based primarily on the DEC Alpha 21064 [McL93], except for the the floating-point pipeline, which is based on the VAX 8600 Model 200 [BBC+89]. We show how the pipeline instantiates the framework parameters and discuss how the framework can be used to guide the verification of the pipeline. Because we have concentrated on developing general techniques for complex pipelines and not on verifying a specific circuit, we do not yet have rigorous proofs for this example.

The floating-point pipeline in the ADK consists of five main stages (AddRecode, MultShift, AddNorm, AddRound, and WriteBack) plus the Divide stage. Most transactions go through each of the main stages once and go through the stages in the same order. The two classes of transactions that are exceptions are: multiplies, which use the MultShift stage twice, and divides, which use the Divide stage between eighteen and fifty-four cycles and skip the MultShift stage.

6.1 Instantiation of Framework Parameters

This pipeline uses a transit protocol scheme. In a transit protocol scheme, when a transaction enters a pipeline it is guaranteed not to encounter any collisions. This pipeline contains a circuit at the entrance that only accepts requests from the environment if it can guarantee that they will not collide with any transaction already in the pipeline. The circuit knows what stages will be available in the future. Davidson's shift-register based circuit [Dav71], which is described in

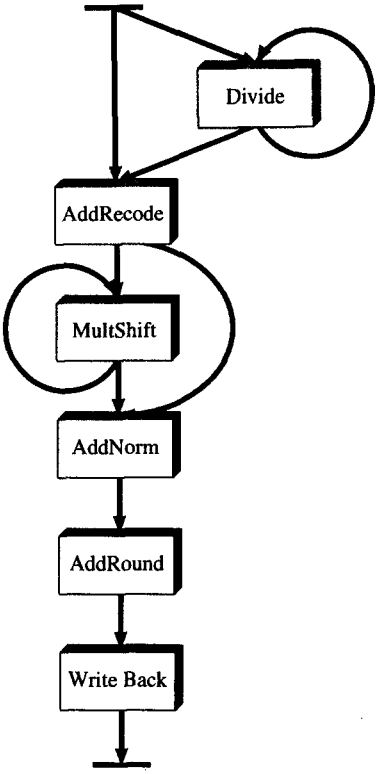

Fig. 4. Block diagram of floating-point pipeline of Adirondack microprocessor

many computer architecture textbooks, is the standard implementation for this protocol scheme.

Because divide transactions have such variable latencies, it would be very inefficient to prevent a transaction from entering the pipeline whenever there is a chance that it may collide with a divide transaction. The Divide stage sends a request to the protocol unit just before a transaction is ready to transfer from the Divide stage to the AddRecode stage. These requests are always accepted and the protocol unit updates its state to ensure that it does not accept any incoming transactions that will collide with the divide transaction.

Because the pipeline uses a transit protocol scheme, we know that there will never be more than one active request per stage, thus the pipeline uses an exclusive arbitration scheme.

For the control scheme, we need to ensure that the control logic will eventually send all transactions to the exit of the pipeline. Most transactions use each of the main stages in the pipeline once, and so follow the no-loops scheme, but multiplies and divides each contain loops, so in the proof we must show that these loops terminate.

The microprocessor *locks* the destination register for each transaction when

its operands are fetched in the instruction pipeline and then unlocks the register when the transaction writes its results to the register file. This is an implementation of a tags-unique ordering scheme, in that only one transaction with a given tag (destination register) is in the pipeline at a time.

6.2 Preparing the Floating-Point Pipeline for Verification

Before we verify the pipeline, we need to define the combinational-representation, specification, matching function, and constraints.

Definition 3: *Combinational representation of ADK floating-point pipeline*

$$
\begin{array}{l}
\mathit{FloatComb}\ T_i\ T_o \mathrel{\hat{=}} \\
\quad \exists T_d,\ T_s,\ T_1,\ T_2,\ T_s,\ T_3,\ T_4\ . \\
\qquad \mathit{case}\ (\mathit{opcode}\ T_i) \\
\qquad\quad \mathit{of}\ \mathit{AddF}\ \Rightarrow \\
\qquad\qquad ((\mathit{AddRecode}\ \mathit{subExps}\ T_i\ T_1)\ \&\ (\mathit{MultShift}\ \mathit{align}\ T_1\ T_2)\ \& \\
\qquad\qquad\ (\mathit{AddNorm}\ \mathit{add}\ T_2\ T_3)\ \&\ (\mathit{AddRound}\ \mathit{round}\ T_3\ T_4)\ \& \\
\qquad\qquad\ (\mathit{WriteBack}\ T_4\ T_o)) \\
\qquad\quad \mid\ \mathit{DivF}\ \Rightarrow \\
\qquad\qquad ((\mathit{varloop}\ 54\ \mathit{decVal}\ \mathit{Divide}\ T_i\ T_d)\ \& \\
\qquad\qquad\ (\mathit{AddRecode}\ \mathit{add}\ T_d\ T_s)\ \&\ (\mathit{AddNorm}\ \mathit{add}\ T_s\ T_3)\ \& \\
\qquad\qquad\ (\mathit{AddRound}\ \mathit{round}\ T_3\ T_4)\ \&\ (\mathit{WriteBack}\ T_4\ T_o)) \\
\qquad\quad \mid\ \mathit{SubF}\ \Rightarrow\ \ldots \\
\qquad\quad \mid\ \mathit{MulF}\ \Rightarrow\ \ldots
\end{array}
$$

The combinational representation of the pipeline (*FloatComb*) is shown in Definition 3, which for conciseness only includes the addition and division operations. We begin by existentially quantifying on the internal signals in the pipeline and then do a case split on the opcode of the input transaction. Each path describes the stages that the transaction goes through and the operation that the stages perform. Addition transactions go through the five main stages, performing the *subExps* operation in the *AddRecode* stage, *etc.*. The path for division transactions includes a data-dependent loop with the *Divide* stage. The *varloop* function initializes the loop counter to fifty-four and decrements it a data-dependent amount (one, two or three — corresponding to the number of bits of the quotient that were calculated) each iteration, until it reaches zero. The amount is calculated by applying the function *decVal* to the transaction in the stage.

The specification for the pipeline (*FloatSpec*, Definition 4) checks that the destination register of the output transaction is the same as that for the input transaction, does a case split on the opcode of the input transaction and then checks that the result (*getResult* T_o) is correct for the input operands (*getIOps* T_i).

Definition 4: *Specification of ADK floating-point pipeline*

$$
\begin{array}{l}
\mathit{FloatSpec}\ T_i\ T_o \triangleq \\
\quad \mathit{getDestReg}\ T_i = \mathit{getDestReg}\ T_o\ \&\ \\
\quad \mathit{case}\ (\mathit{opcode}\ T_i) \\
\quad\quad \mathit{of}\ \mathit{AddF} \Rightarrow \mathit{AddFSpec}\ (\mathit{getIOps}\ T_i)\ (\mathit{getResult}\ T_o) \\
\quad\quad \mid\ \mathit{SubF} \Rightarrow \mathit{SubFSpec}\ (\mathit{getIOps}\ T_i)\ (\mathit{getResult}\ T_o) \\
\quad\quad \mid\ \mathit{MulF} \Rightarrow \mathit{MulFSpec}\ (\mathit{getIOps}\ T_i)\ (\mathit{getResult}\ T_o) \\
\quad\quad \mid\ \mathit{DivF} \Rightarrow \mathit{DivFSpec}\ (\mathit{getIOps}\ T_i)\ (\mathit{getResult}\ T_o)
\end{array}
$$

The pipeline uses a tags-unique ordering scheme, where transactions are tagged with their destination register. Each ordering scheme has an associated generic relation that is used as the basis of the matching relations for pipelines. The generic matching relation for a tags-unique scheme is *tagsUnique*, shown in Definition 5. It is a relation over an input stream and an output stream of tags, requests and accepts. Tags are only significant when a transaction transfers into or out of the pipeline, so we use the request and accept signals to detect transfers into and out of the pipeline.

In Definition 7 the matching relation for the floating-point pipeline (*FloatMatch*) is defined in terms of *tagsUnique* and *mkTagFloat*, which extracts the destination register for a transaction.

Definition 5: *Matching with unique active tags*

$$
\begin{array}{l}
\mathit{tagsUnique}\ I\ O\ t_i\ t_o \triangleq \\
\quad \mathit{let}\ (\mathit{TagP},\ \mathit{ReqP},\ \mathit{AccP}) \triangleq \mathit{map}\ \mathit{split3}\ I \\
\quad\quad (\mathit{TagN},\ \mathit{ReqN},\ \mathit{AccN}) \triangleq \mathit{map}\ \mathit{split3}\ O \\
\quad \mathit{in} \\
\quad\quad (\mathit{ReqP}\ t_i\ \&\ \mathit{AccP}\ t_i)\ \& \\
\quad\quad \mathit{nextTimeTrue}\ (\lambda t\ .\ (\mathit{TagN}\ t = \mathit{TagP}\ t_i)\ \&\ (\mathit{ReqN}\ t)\ \&\ (\mathit{AccN}\ t))\ t_i\ t_o \\
\quad \mathit{end}
\end{array}
$$

Definition 6: *Function to calculate tags for ADK floating-point pipeline*

$$\mathit{mkTagFloat}\ (\mathit{Trans},\ \mathit{Req},\ \mathit{Acc}) \triangleq (\mathit{getDestReg}\ \mathit{Trans},\ \mathit{Req},\ \mathit{Acc})$$

Definition 7: *Matching relation for ADK floating-point pipeline*

$$
\begin{array}{l}
\mathit{FloatMatch}\ I\ O\ t_i\ t_o \triangleq \\
\quad \mathit{tagsUnique}\ (\mathit{map}\ \mathit{mkTagFloat}\ I)\ (\mathit{map}\ \mathit{mkTagFloat}\ O)\ t_i\ t_o
\end{array}
$$

To guarantee that the tags in the pipeline are unique, the pipeline imposes the constraint on the the environment that a transaction does not begin to request the pipeline if the previous transaction with the same tag has not yet exited. This is captured in *FloatConst* (Definition 8), which says that if a transaction enters the pipeline at t_i and exits at t_o, then the next time (t_i') that a transaction with the same tag requests the pipeline must be after t_o.

Definition 8: *Constraints for ADK floating-point pipeline*

$$
\begin{array}{l}
\mathit{FloatConst}\ (I\ \mathit{as}\ (\mathit{TagP},\ \mathit{ReqP},\ \mathit{AccP}))\ O \hat{=} \\
\quad \forall\ t_i,\ t_o\ . \\
\qquad \mathit{FloatMatch}\ I\ O\ t_i\ t_o\ \& \\
\qquad (\mathit{nextTimeTrue} \\
\qquad\quad (\lambda t\ .\ (\mathit{mkTagFloat}\ (I\ t)) = (\mathit{mkTagFloat}\ (I\ t_i))\ \&\ (\mathit{ReqP}\ t)) \\
\qquad\quad (t_i{+}1)\ t_i{}') \implies \\
\qquad\qquad t_o < t_i{}'
\end{array}
$$

6.3 Verification of Floating-Point Pipeline

Following the process described in Section 3 We begin the verification of a pipeline by saying that the implementation, matching relation, and constraints satisfy the specification (Equation 2).

$$(\mathit{FloatPipe},\ \mathit{FloatMatch},\ \mathit{FloatConst}) \sqsupset \mathit{FloatSpec} \tag{2}$$

We use Theorem 1 to introduce the combinational representation of the pipeline (*FloatComb*) and separate the proof into pipelining and functionality goals. We then apply Theorem 2 and simplify the pipelining proof. This leaves us with the five proof goals that were listed in Table 1: the cores of the segments are valid, the constraints of the segments are met, the combinational representation accurately represents the pipeline, the specifications of the framework parameters are met, and the combinational representation implies the specification.

Because the segments in this pipeline are just stages and we derived the combinational representation using the standard algorithm, the first and third goals are easily solved. None of the stages impose any constraints on the pipeline, so the second proof obligation from Table 1 is also trivially solved.

The fourth obligation is to prove that the framework specifications are met. The pipeline uses Davidson's shift-register circuit to maintain the current state of the pipeline and guarantee that transactions do not collide. This is a valid implementation of a transit protocol scheme and so takes care of the protocol specifications. Because we are using a transit protocol scheme, we know that transactions will always be immediately accepted, and so the arbitration specifications are met.

For the control scheme, we need to prove that all paths through the pipeline lead to an exit. Multiply transactions use the MultShift stage exactly twice, and therefore these loops terminate. For divide transactions, we begin with a finite upper bound (fifty-four) on the loop counter for the divide stage and decrement the counter each iteration until we reach zero. For the tags-unique ordering scheme, the pipeline constrains the environment such that transactions do not request to enter the pipeline if the previous transaction with the same tag has not yet exited. We combine this with a lemma that a transaction's tag does not change as it traverses the pipeline and prove that the tag of each transaction in the pipeline is unique.

The fourth and final proof obligation is to show that *FloatComb* implies *FloatSpec*. This proof is still a significant challenge, but it is far simpler than it would have been without the framework. The floating-point datapath is complex and the abstraction gap between the implementation and specification is very large, but the proof is purely combinational and does not need to reason about pipelining concerns.

7 Conclusion

The work presented here is aimed at developing general techniques for applying formal methods to pipelined circuits. We began with a formal definition of correctness for pipelines that guarantees that transactions are able to traverse through the pipeline and will exit with the correct data. This definition allows us to separate the pipelining and functionality related parts of the verification into two proofs. The pipelining proof is data-independent (except for pipelines that use data-dependent control) and the functionality proof is purely combinational. We simplified the pipelining proof by introducing a framework with four parameters (protocol, arbitration, control, and ordering) that characterize pipelines. Each parameter has an associated set of specifications that are used to verify the correctness of the pipelining circuitry. We have found a set of commonly used instantiations of the framework parameters that sufficient to characterize and the guide the verification of a number of commercial microprocessors.

Acknowledgments

We are deeply indebted to the Semiconductor Engineering Group at DEC, who have provided a great deal of useful information and feedback. Mark Aagaard is supported by a fellowship from the Digital Equipment Corporation. Miriam Leeser is supported in part by NSF National Young Investigator Award CCR-9257280.

References

[AA93] D. Alpert and D. Avnon. Architecture of the Pentium microprocessor. *IEEE Micro*, 12:11–21, June 1993.

[AAD+93] T. Aspre, G. Averill, E. DeLano, R. Mason, B. Weiner, and J. Yetter. Performance features of the PA7100 microprocessor. *IEEE Micro*, pages 22–35, June 1993.

[AL93] M. D. Aagaard and M. E. Leeser. A framework for specifying and designing pipelines. In *ICCD*, pages 548–551. IEEE Comp. Soc. Press, Washington D.C., October 1993.

[BBC+89] B. J. Benschneider, W. J. Bowhill, E. M. Cooper, M. N. Gavrelov, P. E. Gronowski, V. K. Maheshwari, V. Peng, J. D. Pickholtz, and S. Samudrala. A pipelined 50-MHz CMOS 64-bit floating-point arithmetic processor. *IEEE Jour. of Solid-State Circuits*, 24(5):1317–1323, October 1989.

[BD94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *CAV*, July 1994.

[Bea93] D. L. Beatty. *A methodology for formal hardware verification, with application to microprocessors.* PhD thesis, Computer Science Dept, Carnegie Mellon Univeristy, 1993.

[BT90] A. Bronstein and C. L. Talcott. Formal verification of pipelines. In L. J. M. Claesen, editor, *Formal VLSI Specification and Synthesis*, pages 349–366. Elsevier, 1990.

[Dav71] E. Davidson. The design and control of pipelined function generators. In *Proceedings 1971 International IEEE Conference on Systems, Networks and Computers*, pages 19–21, January 1971.

[HMC94] E. Harcourt, J. Mauney, and T. Cook. From processor timing specifications to static instruction scheduling. In *Static Analysis Symposium*, September 1994.

[McL93] E. McLellan. Alpha AXP architecture and 21064 processor. *IEEE Micro*, 13(3):36–47, June 1993.

[Mil89] V. Milutinovic. *High Level Language Computer Architecture.* Comp. Sci. Press Inc., 1989.

[Mis90] M. Misra. *IBM RISC System/6000 Technology.* IBM, 1990.

[SB90] M. Srivas and M. Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, pages 52–64, September 1990.

[Seg93] C.-J. Seger. Voss — A formal hardware verification system user's guide. Technical Report 93-45, Dept. of Comp. Sci, Univ. of British Columbia, 1993.

[SGGH92] J. B. Saxe, S. J. Garland, J. V. Guttag, and J. J. Horning. Using transformations and verification in circuit design. In *Designing Correct Circuits, Lyngby 1992*, 1992.

[TK93] S. Tahar and R. Kumar. Implementing a methodology for formally verifying RISC processors in HOL. In J. Joyce and C. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications*, pages 283–296, August 1993.

[TK94] S. Tahar and R. Kumar. Implementation issues for verifying RISC-pipeline conflicts in HOL. In J. Camelleri and T. Melham, editors, *Higher Order Logic Theorem Proving and Its Applications*, August 1994.

[WC94] P. J. Windley and M. Coe. A correctness model for pipelined microprocessors. In *Theorem Provers in Circuit Design.* Springer Verlag; New York, 1994.

[Win91] P. J. Windley. The practical verification of microprocessor designs. In *IEEE COMPCON*, pages 462–467. IEEE Comp. Soc. Press, Washington D.C., February 1991.