A FRAMEWORK FOR THE SPECIFICATION, DESIGN, AND VERIFICATION OF PIPELINES WITH STRUCTURAL HAZARDS

Mark David Aagaard

December 1994

EE-CEG-94-6

School of Electrical Engineering
Cornell University
Ithaca, NY 14853
USA

A FRAMEWORK FOR THE SPECIFICATION, DESIGN, AND VERIFICATION OF PIPELINES WITH STRUCTURAL HAZARDS

Mark David Aagaard*

Abstract

Design features found in state-of-the-art microprocessors, such as super-scalar issue, out-of-order completion, and data-dependent control, lead to pipelined implementations that contain structural hazards. The presence of structural hazards increases the the difficulty of specifying and verifying a pipeline. Specifications are more difficult to write, because it is difficult to predict when a particular transaction will use a given stage, transactions may exit out-of-order, and freedom from deadlock and livelock must be included. The effort to verify a pipeline increases due to the larger number of interactions between transactions and the complexity of the hardware required to support the high-performance features.

We have written a formal definition of correctness for pipelines that ensures that transactions that wish to enter the pipeline are able to do so, transactions that enter the pipeline eventually exit, and output transactions are correct with respect to 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 allows the verification of a pipeline to be divided into two separate tasks: proving that the pipelining circuitry meets the pipeline correctness criteria and proving that the datapath and control circuitry meet the functionality specification. The first proof is almost always data-independent. The second proof is purely combinational: there is no notion of time and each input transaction is dealt with independently.

To further simplify the pipelining proof, we created a framework with four parameters that are used to characterize pipelines. Each parameter has an associated specification. We have proved that pipelines that conform to these specifications obey the pipelining correctness criteria. We analyzed a variety of state-of-the-art commercial microprocessors, including the DEC Alpha 21064, HP PA-Risc 7100, and Intel Pentium, and found a small set of instantiations of the framework parameters that are sufficient to characterize the instruction, integer, and floating-point pipelines in these microprocessors. This set of instantions can be used to informally verify the pipelining aspects of correctness or to guide a formal proof for a design.

*This work was performed while the author was supported by a fellowship from the Digital Equipment Corporation. The author is now with the Department of Computer Science, University of British Columbia, Vancouver BC, Canada (*aagaard@cs.ubc.ca*)

Acknowledgements

My family has always been supportive of me and have helped me maintain a sense of humor while managing all of the various crises involved in surviving graduate school. I'm particularly grateful for their selflessness in devoting many hours over the last few months helping me to type the thesis — allowing my hands to recover from the tendonitis that had developed.

When I entered Cornell I had never heard the words "formal methods of hardware design" or "hardware verification." I had only the vague notion that I enjoyed designing hardware but didn't enjoy debugging prototypes. I have to thank Miriam Leeser for introducing me to this area of research, prompting me to investigate the formal verification of pipelines, and encouraging me to explore the potential of the vague ideas that eventually grew into this thesis.

I am extremely indebted to the Semiconductor Engineering Group at Digital Equipment Corporation, especially Tim Leonard, Mike Kantrowitz, and Will Sherwood. For the past three years I have been financially supported through a generous fellowship from Digital. In many ways though, their cooperation and willingness to share information about microprocessor design and verification in the real world was even more valuable than the financial aspects of the fellowship.

Contents

List of Figures

List of Tables

List of Definitions

List of Theorems

List of Lemmas

List of Specifications

Chapter 1

Introduction

1.1 Introduction

Microprocessors are continuing to increase in complexity; by the end of this decade many super-scalar super-pipelined processors will be issuing more than eight instructions per clock cycle and will contain pipelines ranging from six to ten or more stages in length. This means that is quite possible that fifty or more instructions will be executing simultaneously in the microprocessor. At the same time, design cycles are continuing to shrink, which raises the cost of production delays due to bugs and other causes. Even today, state-of-the-art microprocessors are so complex that engineers are sometimes unwilling to implement proposed designs because current tools and methods are unable to give them sufficient confidence that their implementation would be correct. This can lead engineers to favor incremental changes in existing designs over revolutionary innovations.

The goal of the work presented here is to develop systematic techniques for verifying large-scale, complex pipelined circuits. 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].

We view a pipeline as a network of *stages* through which *transactions* flow. A transaction is an atomic packet containing control information and data. Pipelines operate in *environments*, which are responsible for *issuing* transactions to the pipeline and receiving output transactions from the pipeline. In most pipelines found in modern microprocessors, there is a potential that two or more transactions will want to use the same stage at the same time. This situation is known as a *collision* and the stage that is the source of the collision represents a *structural hazard*.

Structural hazards are present in pipelines that utilize high-performance features such as superscalar issue and data-dependent control. 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 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.

We have defined what it means for a pipeline to be free from structural conflicts*and to conform to a functional specification. This definition guarantees that every transaction that wishes to enter the pipeline is able to do so, every input transaction exits from the pipeline, and every output transaction is correct with respect to the functional specification. We separate the definition 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 proof and a functionality proof. The pipelining proof is data-independent for almost all pipelines and the functionality proof is purely combinational. Without this separation of concerns, every possible sequence of inputs must be considered when verifying a pipeline. By separating the pipelining and functionality aspects of verification and using a combinational functionality proof, the correctness of each operation that the pipeline performs can be verified individually.

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 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. Using the framework divides the proof that a pipeline is free from structural conflicts into separate and largely independent tasks. Because each parameter to the framework has its own specification, proofs can be structured into independent parts. This makes it possible to quickly determine which parts of the proof need to be updated when changes are made to an implementation. One of the key features of the framework is that the parameters can be implemented using mixtures of hardware and constraints on the environment. The

*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.

specifications for the parameters describe necessary *functionality*; only as much hardware as required to correctly produce the required functionality has to be used. There are degenerate cases in which parameters can be implemented without any hardware.

From the beginning we sought to develop general verification techniques that would be applicable to as many pipelines as possible. In particular, the specifications for the parameters are very general, so as to allow for innovative and unforeseen design solutions. There is a tradeoff here, in that increasing the generality of the specifications enlarges the number of pipelines that can be reasoned about, but passes up the opportunity to capture reasoning that may be common to a significant class of pipelines. By examining a variety of commercial microprocessors, we found a relatively small number of protocol, arbitration, control, and ordering schemes that are sufficient to characterize the instruction, integer, and floating-point pipelines in these microprocessors. The list includes the Digital Alpha 21064 [McL93], HP PA-Risc 7100 [AAD+93], IBM RS/6000 [Mis90], IBM PowerPC 601 [B+93], Intel Pentium [AA93], MIPS R4000 [MWV92], and Motorola 88110 [DA92]. Knowing which of these standard schemes a pipeline uses for each parameter helps guide the proof that the pipeline meets the specifications of the framework parameters. Thus, we gain the benefits of a very general set of specifications, but are still able to capture common reasoning.

We developed decision trees that simplify finding which instantiations are valid for a particular pipeline. Some of the questions used in the decision trees are quite easy to answer, (*e.g.* "Does every transaction follow the same path through the pipeline?"), while others may require some justification (*e.g.* "Does the environment issue transactions such that they will never collide in the pipeline?"). The justifications for the non-trivial questions represent a set of obligations that are incurred when using a decision tree. The pipelining aspects of correctness may be informally verified by using the decision trees to choose instantiations for the parameters and then verifying that the pipeline obeys the obligations that were incurred.

Pipeline verification efforts have typically dealt solely with detailed descriptions of specific implementations. This meant that it was infeasible to verify a design until it was complete and changes in the design were difficult to accommodate in the proof. With the framework, the formality of the verification can increase in proportion to the level of detail in the design. Even when a design is nothing more than scribbled boxes on the back of an envelope, many design decisions can be checked against the informal specifications for the parameters. As the design becomes more detailed, the proof can become more detailed and rigorous.

1.2 Related Work

Much of the pioneering work in developing design methodologies for pipelines was done in the early seventies by Davidson and Kogge. Davidson invented the reservation table for describing the flow of transactions through a pipeline and one of the standard circuits for scheduling transactions in pipelines with structural hazards [Dav71]. Kogge derived and verified additional algorithms and later wrote the standard reference book on pipeline design [Kog81]. In the rest of this section we describe work aimed at the formal verification of pipelines. We divide the work into two categories: that which is based on theorem-proving and that based on automated methods, namely model-checking and symbolic simulation. Our primary focus of comparison is how effectively the methods reason about structural hazards and the definition of correctness that is used. We are interested in what structural hazards, if any, they have dealt with; what effect complex features, such as superscalar issue or data-dependent control, would have on the efficacy of the verification techniques; whether they separate pipelining and functionality verification; and if their definition of correctness guarantees that all transactions that enter the pipeline are able to exit and that all transactions that wish to enter the pipeline are able to do so.

1.2.1 Theorem Proving Based Verification of Pipelines

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 toy microprocessor using the Larch Proof Assistant [SGGH92]. The Mini-Cayuga is a three-stage pipeline and Saxe's example has four stages. In each case all transactions follow the same path and spend exactly one cycle in each stage.

Bronstein [BT90] used string-functional semantics [BT89] to reason about pipelines in the Boyer-Moore theorem prover NQTHM [BM88]. The most complicated of these circuits was Saxe's example microprocessor. Signals in string-functional semantics are finite length lists of values. Bronstein defined two relations for verifying a pipelined implementation against a non-pipelined specification: α for pipelines in which all transactions have the same latency and β for stuttering pipelines. A string from a pipeline is a stuttering of a specification string if removing consecutive identical elements from the pipeline string produces the specification string. For example, $aaaxxy$ is a stuttering of axy, but ayx is not. Bronstein's work has been extended by Van Aelten, and Bhagwati is continuing the development of Van Aelten's work. Both Van Aelten and Bhagwati use model-checking based techniques, so we postpone further discussion of their work until Section 1.2.2. The pipelines verified by Saxe, Srivas, and Bronstein do not contain any structural hazards, so proving termination and liveness properties was not a significant concern. Saxe's microprocessor has data hazards, which are dealt with using bypass paths (results are fed back to subsequent transactions without going through the register file) and control hazards, which are dealt with using branch prediction and aborting transactions when a branch is mispredicted.

Tahar is using HOL to verify a simplified version of Hennessy and Patterson's academic microprocessor, the DLX [TK93]. As in the work presented in this thesis, 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. Tahar defines pipelining correctness to be the absence of structural, data, and control conflicts. He then separately verifies that the DLX does not have any conflicts and that the datapath and control circuitry are correct. This does not formally guarantee that transactions that enter the pipeline will exit nor that transactions that wish to enter the pipeline will be able to do so. Tahar's model of a pipeline requires that transactions never stall within the pipeline. This implicitly deals with the pipelining correctness concerns, but restricts the class of pipelines that can be handled to those where verifying these properties would be trivial.

Tahar's definition of a structural conflict is that two transactions attempt to access the same resource at the same time. The inability to describe stalls coupled with this definition of structural conflicts prevents almost all pipelines with structural hazards from being considered. The only pipelines with structural hazards that can be handled are those where the environment is responsible for issuing transactions so that no conflicts occur between transactions. These limitations are reflected in the description of Tahar's version of the DLX. Tahar uses a delayed branch and requires that there is a one clock cycle delay between a transaction that performs a load from memory and a subsequent transaction that uses the result. This exposes pipelining aspects of the implementation in the specification. As defined by Hennessy and Patterson, the DLX uses branch-prediction and and a hardware interlock to hide all aspects of pipelining from the specification.

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.

Windley has used these techniques to verify the Uinta microprocessor, which has a five stage pipeline. There is one source of stalls: data hazards caused by a load instruction followed immediately by a use of the result causes a stall for one cycle in the third stage. The hardware detects a dependency between a load and the subsequent transaction in the second stage. This allows the hardware to insert a no-op transaction in the instruction stream. Only the transaction using the result of the load is stalled, because the following instruction is the no-op rather than the next instruction in the input stream. This stall is relatively simple, yet it adds an extra case-split (stall or no-stall) to the descriptions of three of the five stages and to the abstraction mapping. The abstraction mapping for each pipeline must be derived separately, and Windley has not yet attempted to verify a pipeline with superscalar issue or data-dependent control. Clearly, these features will complicate the process of finding an abstraction mapping and the verification of a pipeline. It is difficult to predict exactly how complex the mappings will become or whether techniques will be found that help automate the process of defining the mapping for a pipeline.

Windley's abstraction mappings encompass structural, data, and control hazards. Termination concerns are handled by mapping the time that a transaction is supposed to modify each element of the microprocessor state to the corresponding modification in the specification. If a transaction does not terminate, then some of the state variables that it is supposed to modify will not be modified. This will result in an inconsistency between the implementation and the specification, which will be caught as part of the normal verification process. Errors due to data and control conflicts are detected in a similar fashion. Abstraction mappings guarantee that once a transaction enters the pipeline, it executes correctly. As part of the proof of correctness, Windley needed to show that Uinta never enters a state where it stalls for forever. This guarantees that every transaction will eventually be able to enter the pipeline.

We verified the framework by proving theorems 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 [C$^+$86], HOL [MG93], and PVS [SOR93]. One of our next goals is to explore mechanizing our proofs. Toward that end, there are two efforts in particular that would likely provide useful bases to build upon. Bainbridge, Camelleri and Fleming have verified a cache system for a microprocessor [BCF92] in HOL. They reasoned about *transactions* as packets of information flowing through a system. They also developed a library of theorems for temporal reasoning and point out the usefulness of transitivity lemmas as a technique for decomposing a verification problem into a number of smaller tasks. The verification of the framework reasons about transactions in a similar style and relies upon several transitivity lemmas.

Fura has used Windley's interpreters technology with transactions to verify a processor interface unit [FWC93] in HOL. This system exhibits pipelined behavior in the form of overlapped computation and much of the general reasoning developed is applicable to pipelined circuits. Fura reasons about events occurring over an interval of time, orderings of events, and one-to-one mappings between events. A significant portion of the utilitarian reasoning in the framework deals with these same concerns.

1.2.2 Automated Verification of Pipelines

Van Aelten [VAAD93] has built upon Bronstein's work with string-functional semantics by redefining the β operator for stuttering equivalence so that it subsumes α (delay equivalence) and adding four additional operators. Bronstein used NQTHM; Van Aelten has developed a model checker that uses the equivalence mapping defined using his operators to check the equivalence of a specification and implementation. Van Aelten's β is the most important of his operators. His β operator takes two strings, one with Boolean values and one that is a string of values. The result of applying β to these two strings is a string consisting of those elements in the string of values that had a *true* in the corresponding position in the Boolean string. An implementation is compared against a specification by applying mapping functions built from the operators to the inputs and outputs of the implementation and specification and then using a model checker to compare the resulting machines for equality.

Van Aelten verified a number of small circuits, including a three-bit two-stage pipelined microprocessor. Bhagwati has built on Van Aelten's work by verifying two simple pipelined microprocessors, the more complex of which has a five stage pipeline, four bit datapath, three ALU operations and one register [BD94a]. Neither microprocessor appears to have any structural hazards. Both microprocessors handle control hazards with a branch delay

slot and data-hazards with bypass paths.
Based upon the limited size of the examples that this technique has been able to handle, it is doubtful that pipelines large enough to contain non-trivial structural hazards could be verified. The requirement that the implementation and specification machines are equivalent guarantees that transactions that enter the pipeline will exit. While neither Van Aelten nor Bhagwati address this issue, it is likely that, because every possible sequence of inputs is tested, they could guarantee that the pipeline will never enter a state where it can not ever receive inputs.
Clarke has used computation time logic (CTL) [CES83] to verify a variety of simple three stage arithmetic pipelines with register files. CTL is a very rich temporal logic and so allows properties such as deadlock and livelock to be checked. This expressiveness takes its toll in computation time though, and so only relatively simple examples have been done.
Bryant and Seger invented the notion of ternary trajectory evaluation [BCJ90], which is a much more restricted logic than CTL and which is based on symbolic simulation rather than model-checking. The only temporal operator in ternary trajectory evaluation is the next-state operator. This means that it is very efficient to verify systems when the temporal mapping from the specification to the implementation is precisely known. When there is a large amount of uncertainty in the temporal mapping, each possible mapping must be considered, leading to large inefficiencies. Seger has used the Voss system, which is based on symbolic trajectory evaluation (an extension of ternary trajectory evaluation) to verify an integer pipeline that is reflective of those in a modern RISC microprocessor [Seg93]. As in the other automated systems, the specification says when each input will exit, so termination is guaranteed. This is easy to do for the integer pipeline, because it does not contain any structural hazards. The specification says that a new transaction is received by the pipeline on every cycle that the environment wishes to issue a transaction. The simplicity of symbolic trajectory evaluation should allow more complicated pipelines to be verified, as long as the latency through the pipelines can be accurately predicted.
Beatty has developed the theory of *marked strings*, which provides a basis for the automated verification of circuits, including pipelines [Bea93]. The theory of marked strings is aimed at describing short sequences of overlapped computations. Marked strings provide a way to map a specification downwards to an implementation. The downward mapping allows nondeterminism in the implementation. A string represents a sequence of values. The marks are delimiters used to define a mapping to compare two strings. This allows a string from a detailed, pipelined implementation to be shown to implement an abstract, non-pipelined specification. Once the specification, implementation, and a mapping between the two have been defined, an automated tool based on ternary trajectory evaluation is used.
Beatty has used his system to verify portions of the Hector microprocessor [MBB+86]. In Beatty's words, Hector is "slightly pipelined." Hector does not contain any structural hazards and the main challenges arose from the fact that Hector was a pre-existing microprocessor that had not been designed with formal verification in mind. Beatty verified important properties about several instructions, the initialization sequence and interrupt handling. It is not clear though, how well his methodology will scale to large designs with

complex features such as wide datapaths, superscalar issue, and long latency instructions. Because the pipelining aspects of Hector are relatively simple, termination was assured by the mapping between the implementation and specification and liveness was not a significant concern.

The automated methods described so far have dealt with pipelining and functionality aspects of correctness simultaneously, which has severely restricted the size of pipelines that they can handle. The limiting factor in using model-checking and symbolic simulation techniques to verify a pipeline is the size of the state-space. To mitigate this, Burch [BD94b] has separated the pipelining concerns from the functionality concerns. His work provides a highly automated approach for verifying control and pipelining circuitry, but he does not prove that composing pipelining and functionality correctness results leads to an overall statement of correctness for a pipeline. Burch uses a quantifier-free logic of uninterpreted functions and predicates with equality and propositional connectives.

The first step in Burch's verification process is to build symbolic functions describing the execution of one time step for the pipeline and for the specification. From the function for the pipeline, a second function is derived which describes issuing one transaction to the pipeline and then flushing the pipeline. Flushing a pipeline means to let it run without issuing any new transactions until all transactions in the pipeline have exited. The implementation is valid if, for any initial state and any input transaction, issuing the transaction and then flushing the pipeline is equivalent to executing one time step in the specification. The equivalence is checked by manipulating the two transition functions. This is done using a decision procedure that is similar to a tautology checker and a function for simplifying expressions in the logic of uninterpreted functions with equality. In addition, Burch includes a liveness check that ensures that the pipeline does not enter a state where it could remain stalled forever.

Burch has verified the control and pipelining circuitry for a subset of the DLX. Burch's version of the DLX, like Windley's Uinta, stalls for one clock cycle after a load from memory if the immediately following instruction uses the result of the load. This is handled by using an additional clock cycle to flush the pipeline. This technique of flushing the pipeline and then comparing the implementation and specification is able to handle stalls, but there is a danger that the size of the function describing the implementation will grow too quickly as the length of time required to flush the pipeline increases. Another potential difficulty is that superscalar pipelines may increase the size of the functions beyond that which can be checked efficiently. This is because the function describing the pipeline must describe the simultaneous execution of twice as many transactions as in a single-issue pipeline.

Almost all automated systems require reasoning about known numbers of objects, whether it be the width of a datapath or the number of stages in a pipeline. This prevents the use of induction, one of the most powerful theorem proving techniques. One exception to this is the work by Kurshan and McMillan, who have proved a structural induction theorem for unbounded numbers of processes [KM89]. The theorem requires defining an invariant, process shift operator, and partial-order relation. The method proceeds by proving that if the invariant holds for a system, then using the shift operator to increase the number of processes by one preserves the invariant. By carefully defining the invariant, shift operator, and partial order, the induction proof can be automatically checked using state-space exploration. Although we have taken a very different approach in the development of the framework, there are enough similarities between our formalization of pipelines and the structural induction theorem to warrant investigating whether it could provide an alternative formalism in which to verify the framework.

Kurshan and McMillan have verified the dining philosophers problem, which demonstrates the use of their techniques with a ring of processes and their ability to prove liveness properties. They have also reasoned about the coherency of distributed data in a fully connected network of processes. For pipelines, both safety and liveness properties are critical. We need to ensure that every transaction that enters a pipeline will exit and that every transaction that exits has the correct data. While pipelines are neither rings nor fully connected, they exhibit behavior that is reflective of both topologies. Arithmetic pipelines that support divide or square-root operations contain paths with loops (transactions reuse a stage or set of stages multiple times). In our formalization of pipelines, we treat them as fully connected networks of stages and perform the mapping from a real pipeline to our abstract model by defining the signals between unconnected segments to be always false.

The structural induction theorem requires identical processes, but allows for non-determinism. In our formalization, the signals between stages must conform to a set of behavioral specifications that are the same for each stage. This is compatible with Kurshan's view, in that all of the stages conform to the specifications. Non-determinism can be used to capture the individuality of each stage.

1.2.3 Structural Hazards

A number of the efforts listed here have dealt with control and/or data hazards, but the only pipelines that contain structural hazards are Windley's Uinta and Burch's version of the DLX. In each case the hazard results from a load from memory followed immediately by a use of the result. Both microprocessors resolve the hazard by stalling the pipeline for one clock cycle. This is an example of a very simple structural hazard: the transaction stalls for exactly one clock cycle; there is no contention for resources between multiple transactions (because the stall prevents the issuing of the transaction that would conflict with the transaction waiting for the result); and, except for this one case, all transactions have the same latency through the pipeline. The only other formal work we are aware of that deals with structural hazards in general is Harcourt's use of SCCS [Mil83] 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 Description of Thesis

This thesis is divided into three main parts: Chapters 2 and 3 give an informal description of our definition of correctness for pipelines and the four parameters to the framework; Chapters 4 – 6 describe the use of the framework in the specification, design, and informal verification of pipelines; and Chapters 7 and 8 provide the formal foundations of the framework. Each chapter builds upon the previous ones by presenting more details, increasing the level of formality, or by demonstrating the use of concepts introduced earlier.

In Chapter 2 we give an overview of our model of pipelines and the terminology we use. We then briefly introduce our definition of correctness, the four framework parameters and outline the process of verifying a pipeline. Chapter 3 describes the parameters to the framework and their informal specifications. We illustrate the generality of the framework by showing how a variety of pipelines from commercial microprocessors instantiate the parameters.

We describe how to use the framework as an informal guide to specify, design, and verify pipelines in Chapter 4. We then carry out an extended example for a simple pipeline in Chapter 5. To illustrate the use of the framework with more realistic pipelines, Chapter 6 outlines the development of a fictitious super-scalar pipelined microprocessor, the Adirondack.

The framework is discussed in its full formality in Chapters 7 and 8. Our formal definition of correctness is presented in detail in Chapter 7. This is followed by the two theorems used to decompose the verification of a pipeline into separate tasks and a discussion of these tasks. The key part of verifying the pipelining correctness of an implementation is showing that it conforms to the specifications of the framework parameters. The chapter concludes with a listing of the formal specifications. The first of the two theorems used in the framework allows us to separate the pipelining and functionality aspects of verification. It is very easy to prove. The second theorem says that the specifications of the framework parameters can be used to simplify the pipelining aspects of verification. The proof of this theorem is much more difficult and is the focus of Chapter 8.

We summarize our results and point out areas further research in Chapter 9. Appendix A provides a description of our notation. Appendix B highlights some of the interesting aspects of the formal verification of the example pipeline from Chapter 5 and Appendix C illustrates one aspect of the verification of another pipeline. Sample proofs from Chapter 8 are presented in more detail in Appendix D.

Chapter 2

Overview of Pipelines

2.1 Introduction

This chapter gives an overview of how we describe and reason about pipelines and introduces the framework. We classify the *topology* and behavior of different pipelines and describe how these affect design issues related to the pipeline. We begin with very simple pipelines and work our way up to more complex designs.
We view a pipeline as consisting of a set of stages and existing in an *environment*. The environment issues *transactions* to the pipeline. Inside the pipeline, transactions flow from stage to stage in a synchronous fashion. That is, all transfers of transactions between stages take place on boundaries of clock cycles. At each stage that a transaction passes through, some operation is performed on the data in the transaction. Transactions eventually exit the pipeline, where they are received by the environment.
We divide pipelines into two major classes. In Section 2.2 we describe pipelines where transaction exit the pipeline in the same order that they enter and in Section 2.3 we describe pipelines where transactions may exit out-of-order. Much of the information in Sections 2.2 and 2.3 is loosely based on material by Kogge [Kog81]. These sections do not present any new material, but instead strive to introduce pipelines and their design considerations in a way that will naturally lead into the discussion of the framework.
Section 2.4 contains a brief outline of how we define correctness for pipelines and how the framework guides the verification process. The framework itself is introduced in Section 2.5.

2.2 Pipelines with In-Order Termination

In this section, we describe pipelines in which transactions exit in the same order that they enter. Section 2.3 describes pipelines where transactions may exit out-of-order.

2.2.1 Uniform Unifunctional Pipelines

In the simplest class of pipelines:

- All transactions go through all of the stages in the same order
- Each transaction spends the same amount of time in each stage
- Each stage performs the same operation on all transactions
- The environment issues a transaction to the pipeline every clock cycle
- The environment is always able to receive exiting transactions
- There is only one entrance and one exit port

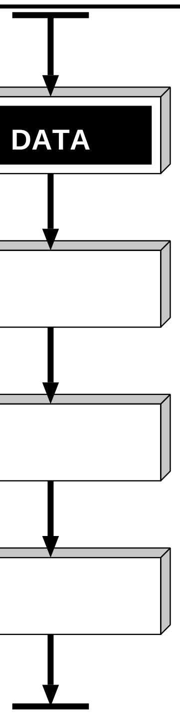


Figure 2.1: Uniform pipe

To include one of these pipelines in an environment, we only need to provide mechanisms that issue transactions to the pipeline and receive the resulting transactions from the output port of the pipeline. No control information needs to be given to the pipeline and transactions consist solely of the data to be operated on. This style of pipeline is typically used in DSP applications, systolic arrays, and some simple instruction fetch/decode pipelines in microprocessors.

As shown in Figure 2.2, in an n-stage pipeline all transactions have a *latency* of exactly n time cycles. That is, transactions exit exactly n cycles after they are issued. Combining this information with the constraint that a new transaction is issued every cycle allows us to deduce that a transaction will exit every cycle.

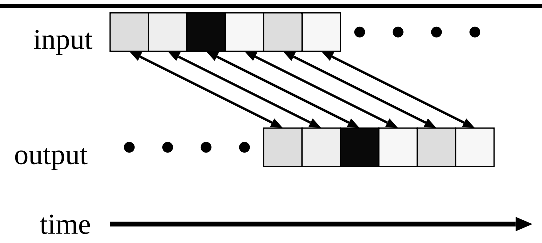


Figure 2.2: A transaction exits every cycle

2.2.2 Multifunctional Uniform Pipelines

The first step up in complexity is reached by allowing stages to perform multiple operations. The pipeline is still uniform, in that all transactions still go through all of the stages in the same order, but different transactions require different operations from the stages. Pipelines that fall into this class include single-issue (non-superscalar) instruction pipelines and integer pipelines in microprocessors. In almost any realistic situation we must also deal with environments which do not issue new transactions every cycle; this is dealt with in Section 2.2.3.

Because each stage can perform multiple operations, we need to supply control information to the stages. We supply this information using data-stationary control (Figure 2.3). Data-stationary control is control information that is contained as part of a transaction.

2.2.3 Detecting Transactions Exiting the Pipeline

If we relax the restriction that transactions have to be issued to the pipeline every cycle, we introduce the possibility that there may be cycles in which there will not be a transaction exiting. In almost any realistic pipeline we will have to confront this situation. This means that the environment needs to have a mechanism for determining when a transaction is exiting. Without such a mechanism, the environment cannot differentiate between a legitimate transaction exiting and garbage data in the last stage. Treating garbage data as a real transaction will likely lead to erroneous behavior. Likewise, if the environment ignores a legitimate transaction exiting the pipeline, errors will also occur. In the rest of the pipelines described in this section and Section 2.3 we need to detect when a transaction is exiting the pipeline. The techniques described in the following paragraph are sufficient for pipelines with in-order termination.

In many cases the environment will know when it is issuing a transaction and each transaction will spend exactly the same number of cycles in the pipeline (n cycles for an n stage pipeline, Figure 2.4). In these cases the environment can calculate when a transaction will be ready to leave. Alternatively, we can make it the pipeline's responsibility to alert the environment when a transaction is exiting. A simple scheme is to use a one-bit `done` signal on the output port. This signal is held high during a cycle that a transaction is exiting.

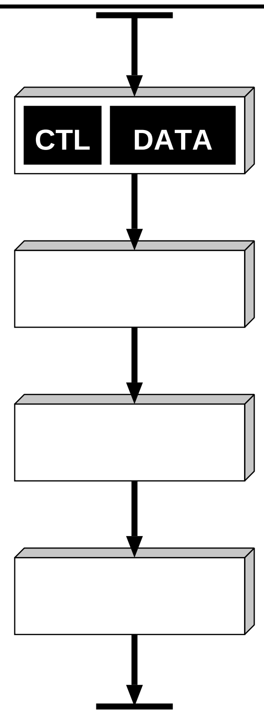


Figure 2.3: Uniform multifunctional pipeline

Another scheme is to use data-stationary control and assign one opcode (NOP) to represent an empty transaction.

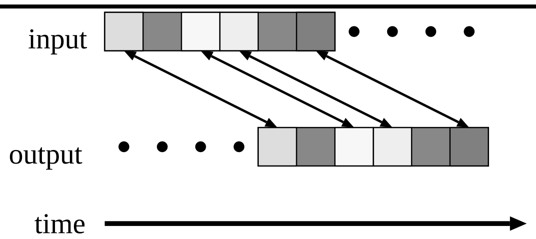


Figure 2.4: Transactions exit after n cycles

2.2.4 Environment May Not Always Be Able to Receive Transactions

We now relax the restriction that the environment has to be always ready to receive transactions. This means that we may have a transaction wishing to exit to the environment when the environment is unable to receive it immediately. This situation is an example of a *collision* and a pipeline where this can happen is said to contain a *structural hazard*. Instruction pipelines for most microprocessors must deal with the fact that the execute pipelines that they send transactions to may be busy and so be unable to receive a transaction immediately.

We can either prevent the collision from happening or we can provide mechanisms to handle it. In the first method we do not allow transactions to enter a pipeline if the environment may be unable to receive them as soon as they are ready to exit. In the second method transactions that are unable to exit when they first wish are *stalled* until the environment is ready. When a transaction is stalled it remains in its current stage until the stage that it wishes to transfer to (or the environment) is able to receive it.

Stalling a transaction in the pipeline might also cause subsequent transactions to stall. This chain of stalls can percolate all the way back to the top of the pipeline, thereby preventing the environment from issuing transactions. If we allow transactions to stall, then we usually have the hardware available to let the environment know when the pipeline can not receive a transaction. Alternatively, we can require the environment to keep track of the state of the pipeline and not issue a transaction when the pipeline is full.

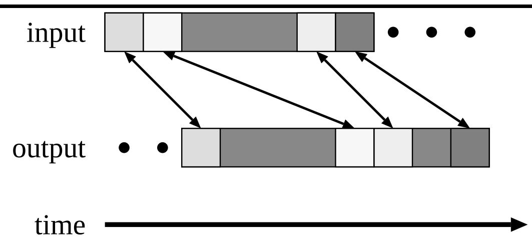


Figure 2.5: Different transactions have different latencies

If transactions may stall, all transactions no longer have the same latency. That is, they do not all spend the same number of clock cycles inside the pipeline; but, transactions still exit in-order. Thus we can use either a done signal or a NOP opcode to determine when a transaction is leaving.

2.2.5 Transactions May Reuse Individual Stages

Control information can be used to specify not only the operation for a stage to perform, but also which stage to send a transaction to next. This allows for different transactions to follow different paths through the pipeline (*nonuniform pipelines*). The information to determine which stage to send a transaction to can come from data-stationary control (as with determining the operation that a stage is to perform (Section 2.2.2)), or *data-dependent* control. Data dependent control is information based upon the result of computation performed by the stage.

The simplest topology that has different paths through a pipeline is one where transactions may reuse individual stages multiple times (Figure 2.6). The DEC Alpha 21064 [McL93], the HP 7100 [AAD+93], and the MIPS R4000 [MWV92] floating-point pipelines reuse a single stage.

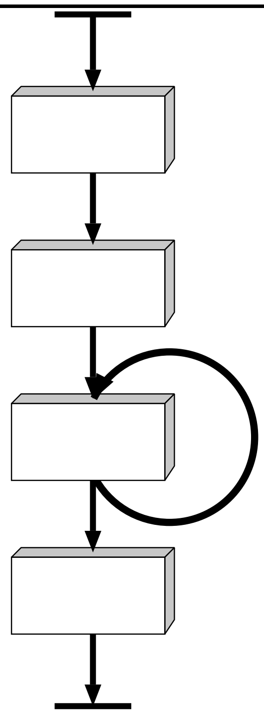


Figure 2.6: Pipeline with self-loop

Just as with pipelines in which the environment may not always be able to receive transactions, pipelines with self-looping stages contain structural hazards. We can handle the hazards by either preventing transactions from entering the pipeline if they might collide with another transaction, or by allowing transactions to proceed through the pipeline until just before a collision.

If we choose the second option, then we have to prevent collisions caused by two transactions wishing to transfer into the same stage. We know that one of the transactions will be trying to reuse the stage that it is currently in and the other transaction will be trying to transfer into a new stage. We need to stall one of the transactions and let the other proceed. We can't stall the transaction that is reusing its stage, because if it stalls we need to store it in its current stage. Thus, we let the first transaction reuse its stage, and stall the second transaction.

Transactions that reuse stages are another cause of different transactions having different latencies. Transactions will leave the pipeline in-order, so we can continue to use either a done signal or a NOP to identify when transactions are leaving the pipeline.

2.3 Pipelines with Out-of-Order Termination

As long as all transactions use all of the stages in the same order (as in Section 2.2), they will exit the pipeline in the same order that they entered. In this section we consider pipelines in which transactions may exit in different orders than they entered (*out-of-order termination*).

In the previously discussed classes of pipelines, we needed to know *when* a transaction was leaving, but it was trivial to determine *which* transaction was leaving. Now we must be able to determine not only when a transaction is exiting, but also which transaction is exiting (Figure 2.7).

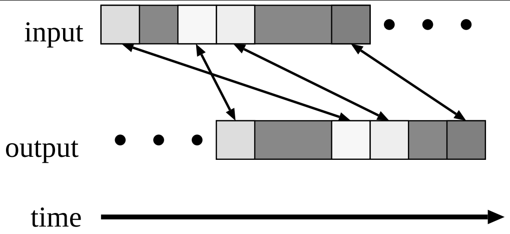


Figure 2.7: Transaction exit pipeline out-of-order

This idea of matching output transactions with input transactions is not necessarily relevant to the operation of the pipeline, but is crucial for verification. In order to verify that a pipeline is correct, one of the things we need to check is that every output transaction contains the correct data. For example, in a microprocessor we need to verify that all ADD instructions produce results that are the sum of the two operands. To check this we need to be able to match up an output transaction with its corresponding input transaction (*e.g.* we need to know the values of the operands to verify that the sum is correct).

To match up a transaction exiting the pipeline with the input transaction that caused it, we usually find a method to assign *tags* to transactions in such a way that transactions with the same tag have the same latency through the pipeline (Figure 2.8), or exit in-order (Figure 2.9). Common sources of tags include the destination of an operation (*e.g.* a location in a register-file or memory) or the class of the transaction (*e.g.* arithmetic or branch).

2.3.1 Transactions Use Different Stages

The first constraint we relax that leads to transactions exiting out of order is to allow different transactions to use different stages. If not all paths have the same latency, then transactions following a long path may be passed by transactions taking shorter paths. For example, in Figure 2.10, transactions which loop in the middle stage may be passed by transactions which skip that stage altogether.

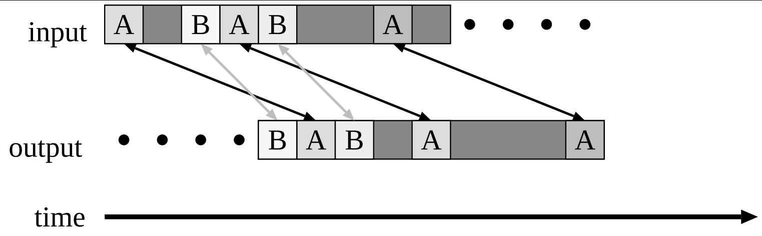


Figure 2.8: Transactions with same tag have same latency

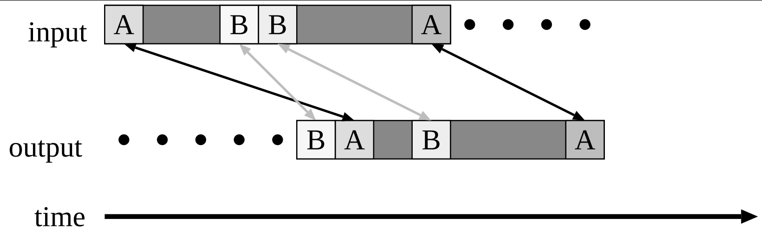


Figure 2.9: Transactions with same tag exit pipeline in-order

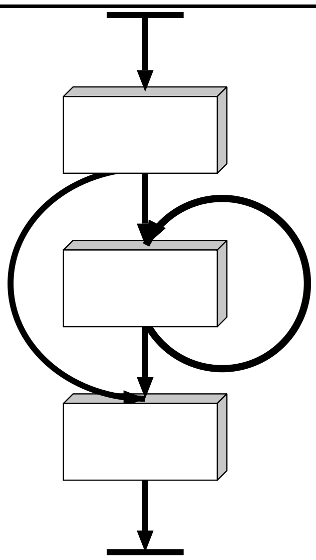


Figure 2.10: Transactions use different stages in pipeline

When transactions follow different paths, dealing with structural hazards becomes more complicated. As with previously discussed classes of pipelines with structural hazards (Sections 2.2.4 and 2.2.5) in which transactions may use a single stage for multiple cycles, we have two choices: either prevent transactions from entering the pipeline if they will collide with another transaction, or stall transactions just before they collide. If we allow transactions to stall, we need to provide methods to choosing which transactions to stall and which to let proceed. If we stall the wrong transaction we can encounter problems with deadlock and livelock. Deadlock occurs when no transaction can proceed to the stage that it wishes to transfer to. In livelock some transactions are able to proceed, but there is at least one transaction that will never be able to proceed.

2.3.2 Pipelines with Multiple Ports

Super-scalar pipelines have multiple entry and exit ports (Figure 2.11). This allows more than one transaction to enter or leave the pipeline at the same time. Once we allow transactions to follow different paths, adding multiple ports causes very little additional complexity. The only significant change is that we can no longer identify transactions solely by the time that they enter or leave the pipeline; we need to specify the time and the port that was used.

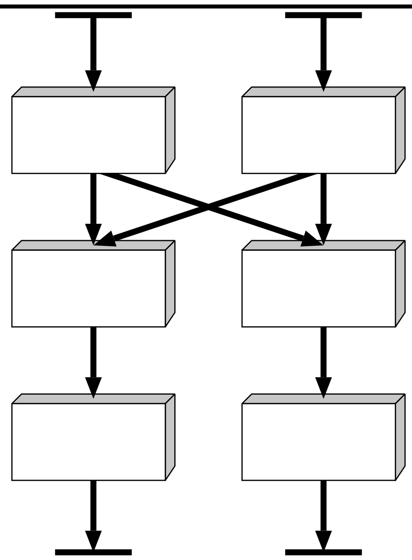


Figure 2.11: Pipeline with two input and two output ports

2.3.3 Structural Hierarchy and Pipelines

When building large or complex pipelines, we often wish to make use of structural abstraction and allow stages to be pipelines. This means that multiple transactions may be inside a single stage. For example, when working with a complete microprocessor we will want to treat the instruction, integer, floating-point, and load/store pipelines as sub-pipelines, which means that they will appear as stages in the top level description. Then we treat each one as an independent pipeline in lower levels of the hierarchy. Hierarchical pipelines introduce a number of complications, because we can no longer characterize the topology of a pipeline solely on the topology of the top level of the hierarchy. For example, in Figure 2.12 all transactions follow the same path through the pipeline at the top level, but transactions may exit the middle stage in a different order than they enter. This means that transactions may exit in a different order despite the fact that the top level appears to be a uniform pipeline.

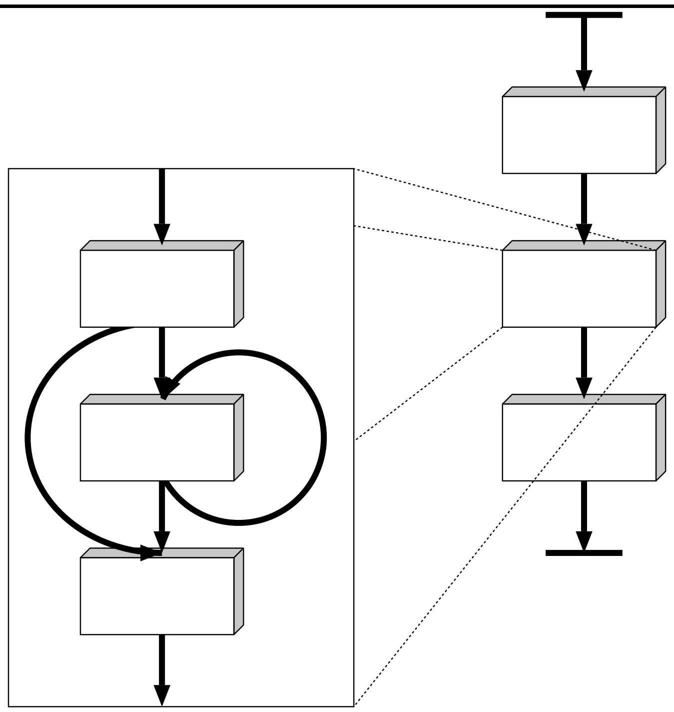


Figure 2.12: Pipeline with structural hierarchy

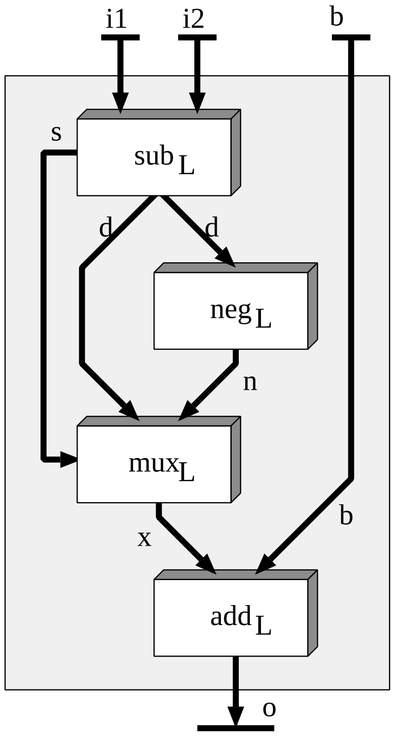
2.4 Verifying Pipelines

This section gives an overview of how we define correctness for a pipeline and how we use the framework to verify pipelines. The purpose is put the description of the framework within a meaningful context. More detailed descriptions of the definitions shown here appear in Chapter 8.

We step through a simple example that calculates $|i_1 - i_2| + b$. This is an extremely simple example, but it contains a structural hazard and makes use of data-dependent control. Chapter 5 describes this example in more detail. The specification for the circuit ($diffAdd_S$) is shown in Equation 2.1

$$diffAdd_S\ (i_1,\ i_2,\ b)\ o\ \hat{=}\ \ o = |i_1 - i_2| + b \tag{2.1}$$

Our combinational logic implementation of the circuit *diffAdd* is shown in Figure 2.13. The variable s is the sign of the result of subtracting i_2 from i_1. The variable d is the result of the subtraction. We negate d using the neg_L circuit and assign the result to n. We then use a multiplexer (mux_L) to set x (the absolute value of $i_1 - i_2$) to be either d or n based on the value of s. We compute the sum of x and b in add_L and generate the output o.



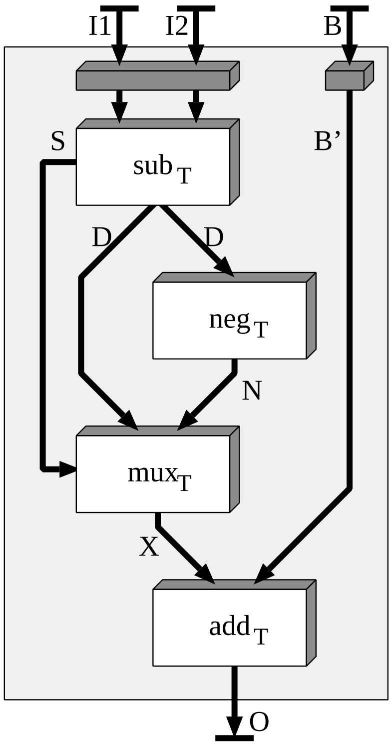
$$\begin{array}{l} diffAdd_L\ (i_1,\ i_2,\ b)\ o\ \hat{=} \\ \exists s,\ d,\ n,\ x\ . \\ \quad sub_L\ (i_1, i2)\ (b,\ d)\ \& \\ \quad neg_L\ d\ n\ \& \\ \quad mux_L\ s\ (d,\ n)\ x\ \& \\ \quad add_L\ (x,\ b)\ o \end{array}$$

Figure 2.13: Combinational logic implementation of diffAdd

Equation 2.2 is the goal for verifying the correctness of *diffAdd*. It says that for all inputs (i_1, i_2 and b) and output (o) the implementation ($diffAdd_L$) implies the specification ($diffAdd_S$).

$$\vdash \forall\ i_1,\ i_2,\ b,\ o.\ diffAdd_L\ (i_1,\ i_2,\ b)\ o \implies diffAdd_S\ (i_1,\ i_2,\ b)\ o \tag{2.2}$$

We add a notion of time to our model and include a register at the top of the circuit to create the implementation shown in Figure 2.14. In this implementation we have capitalized the names of the variables and subscripted circuit components with T to signify that we are now reasoning about sequential circuits (The "T" is for "time").



$reg_T\ B\ B' \mathrel{\hat{=}} \forall\ t\ .\ (B'\ (t{+}1)) = (B\ t)$

$sub_T\ (I_1,\ I_2)\ (S,\ D) \mathrel{\hat{=}}$
$\forall\ t\ .\ sub\ ((I_1\ t),\ (I_2\ t))\ ((S\ t),\ (D\ t))$

$neg_T\ D\ N \mathrel{\hat{=}}$
$\forall\ t\ .\ neg\ (D\ t)\ (N\ t)$

$mux_T\ S\ (D,\ N)\ X \mathrel{\hat{=}}$
$\forall\ t\ .\ mux\ (S\ t)\ ((D\ t),\ (N\ t))\ (X\ t)$

$sub_P\ (I_1,\ I_2)\ (S\ ,D) \mathrel{\hat{=}}$
$\exists I_1{}',\ I_2{}'\ .$
$\quad reg_T\ (I_1,\ I_2)\ (I_1{}',\ I_2{}')$ &
$\quad sub_T\ (I_1{}',\ I_2{}')\ (S,\ D)$

$diffAdd_T\ (I_1,\ I_2,\ B)\ O \mathrel{\hat{=}}$
$\exists B',\ S,\ D,\ N,\ X\ .$
$\quad reg_T\ B\ B'$ &
$\quad reg_T\ (I_1,\ I_2)\ (I_1{}',\ I_2{}')$ &
$\quad sub_T\ (I_1{}',\ I_2{}')\ (S,\ D)$ &
$\quad neg_T\ D\ N$ &
$\quad mux_T\ S\ (D,\ N)\ X$ &
$\quad add_T\ (X,\ B')\ O$

Figure 2.14: Implementation of diffAdd with register

The proof goal for the implementation with registers on the inputs is shown in Equation 2.3. We continue to use the same specification ($diffAdd_S$) but we sample the input and output signals to pass values to $diffAdd_S$. Because the registers delay the signals by one clock cycle, we know that the output at time t+1 should correspond to the inputs at time t.

$$\begin{array}{l} \vdash \forall\ I_1,\ I_2,\ B,\ O\ . \\ \quad diffAdd_T\ (I_1,\ I_2,\ B)\ O \implies \\ \qquad \forall\ t\ .\ diffAdd_S\ ((I_1\ t),\ (I_2\ t),\ (B\ t))\ (O\ (t{+}1)) \end{array} \tag{2.3}$$

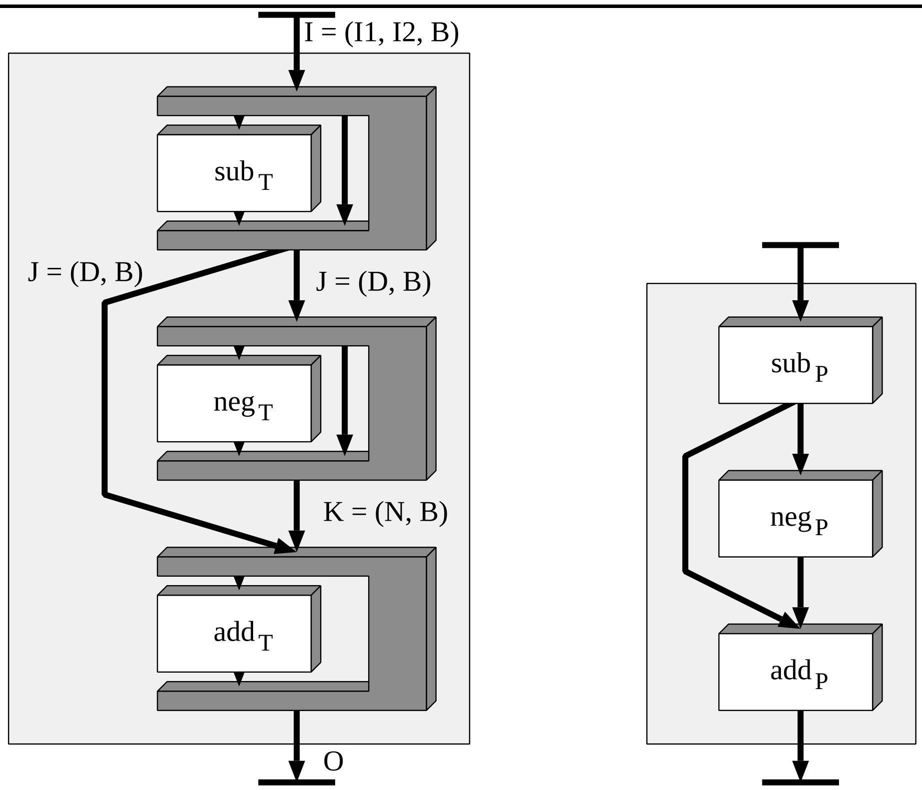


Figure 2.15: Pipelined implementation of diffAdd

We now build the pipelined implementation ($diffAdd_P$) shown in Figure 2.15. We create a pipeline stage for each of the three datapath components (sub_T, neg_T, and add_T) and group the data signals together into transactions that are transferred between the stages. The functionality provided by the multiplexer in the sequential implementation is integrated into the control logic for the stages in the pipeline. We create a stage by taking the corresponding sequential component and adding registers, *control logic* to route transactions through the pipeline, and *pipelining logic* to interface the stage with the other stages in the pipeline. For sub_P and neg_P we also include the signal to store the value of B for the transaction.

There are two paths through the pipeline. If the result of the subtraction in the sub_P stage is positive, then we do not need to negate the result, and so we bypass the neg_P stage. If we do negate the result, then the transaction passes through neg_P before going to the add_P stage. These two paths are shown in Figure 2.16.

In Figure 2.16 and throughout the rest of this section, we will use black and gray tokens to represent transactions flowing through the pipeline. Black tokens are able to bypass the negate stage and gray tokens use the negate stage. Below each block diagram of the pipeline is a reservation table showing the corresponding path of transactions through the pipeline. The vertical axis shows the input port of the pipeline, the stages in the pipeline, and the output port. The horizontal axis represents the progression of time.

An attempt at a proof goal for the pipelined implementation is shown in Equation 2.4. The problem with this goal is that we have not said at what time the output signal should correspond to the input signals.

$$\begin{aligned} &\vdash \forall\ I : \mathit{int\ \ signal}\ .\\ &\qquad \mathit{diffAdd}_P\ I\ O \implies \forall\ t\ .\ \mathit{diffAdd}_S\ (I\ t))\ (O\ (\boxed{t{+}?})) \end{aligned} \tag{2.4}$$

Before we can answer the question of how to match up output transactions with their corresponding input transactions, we need to further examine how transactions flow through the pipeline. If there is a transaction in the sub_P stage that wishes to bypass the neg_P stage at the same time that there is a transaction in neg_P, then we will have a collision because both transactions want to transfer into the add_P stage (Figure 2.17). This collision is an example of a structural hazard. To resolve this hazard we need to define mechanisms for sub_P and neg_P to notify add_P when they have transactions that they wish to transfer; add_P must choose which transaction to accept and notify sub_P and neg_P of its decision; and the stage which cannot transfer its transaction must stall until add_P is able to receive the transaction.

The framework provides support for identifying structural hazards in pipelines and then choosing and verifying the mechanisms needed to resolve the hazards. In this pipeline we force a transaction in sub_P that wants to transfer to add_P to stall if there is a transaction in neg_P. Chapter 5 describes in more detail how the framework guides the process of recognizing and resolving structural hazards for this example.

Having chosen to stall transactions inside the pipeline to avoid collisions, we return to the question of verifying the pipeline. We may be tempted to write the proof goal as shown in Equation 2.5. This says that for all times (t) the output at time t+3 will correspond to the current inputs.

$$\forall\ I,\ O\ .\ \mathit{diffAdd}_P\ I\ O \implies \forall\ \boxed{t}\ .\ \mathit{diffAdd}_S\ (I\ t)\ (O(\boxed{t{+}3})) \tag{2.5}$$

As demonstrated by Figure 2.18, this goal has two problems. First, the environment may not issue a new transaction every cycle. Thus, we cannot say that "*for all times t*" the implementation implies the specification. Second, not every transactions spends exactly three cycles in the pipeline, so the output at time t+3 may not correspond to the input at time t.

By forcing transactions to stall in sub_P, we cause transactions exit the pipeline in the same order that they enter (Figure 2.19). Because transactions exit in-order, we know that the n^{th} transaction to exit the pipeline corresponds to the n^{th} transaction that entered the pipeline. We use the relation *matchInOrder* (Definition 4.4 in Chapter 4) over an input

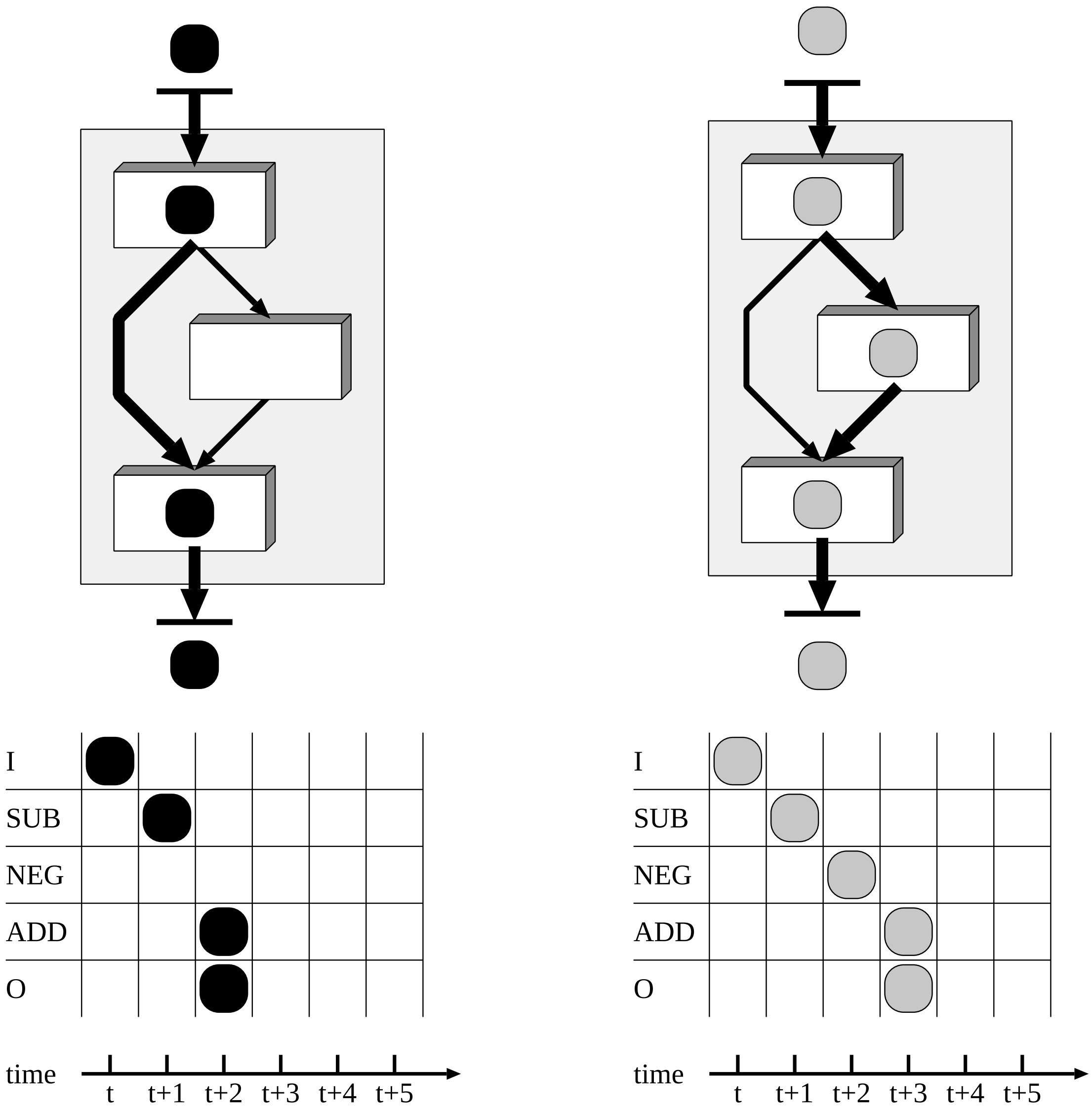
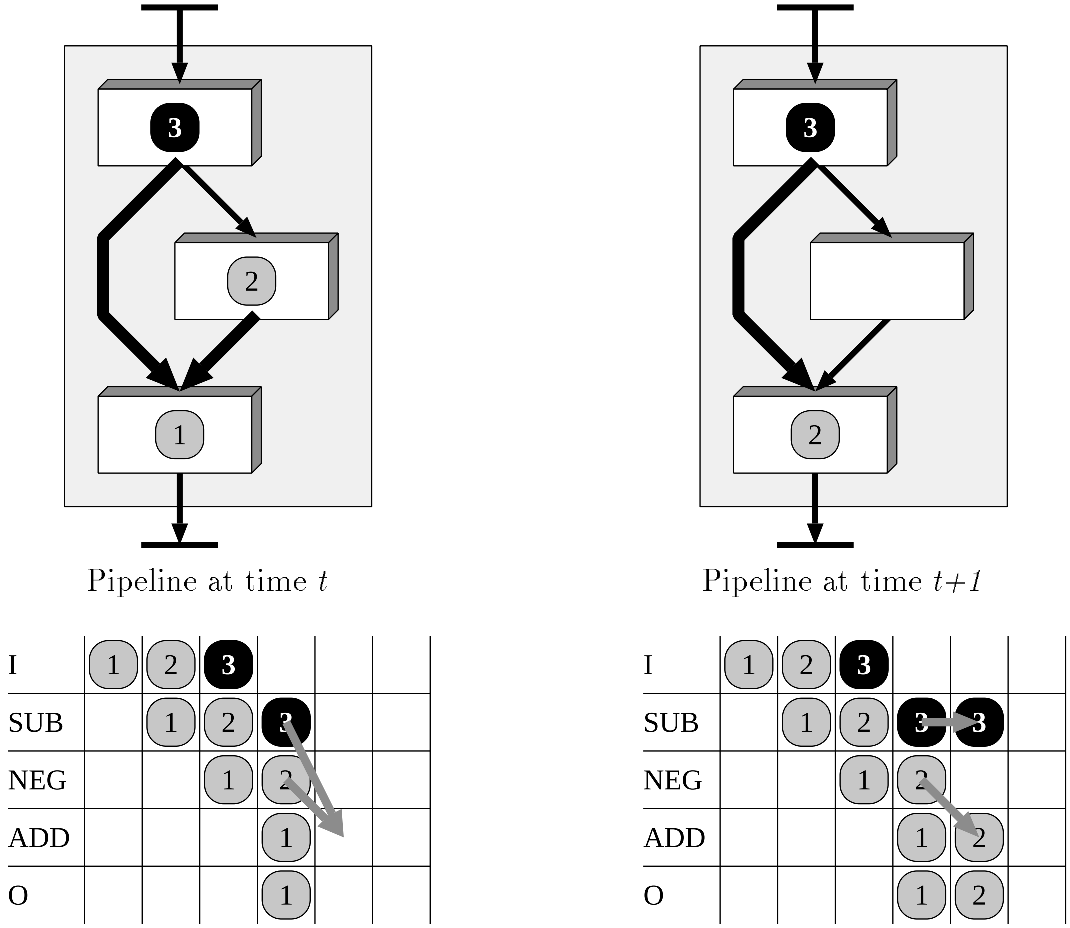


Figure 2.16: Two possible paths through the pipeline



Pipeline at time *t* — Pipeline at time *t+1*

Transactions will collide — Stall transaction in *sub*

Figure 2.17: Stall transaction in *sub* to prevent collision

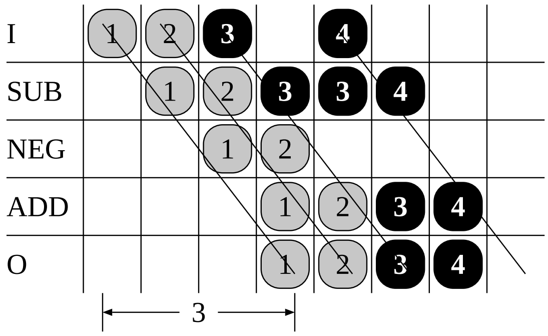


Figure 2.18: Flow of transactions through pipeline

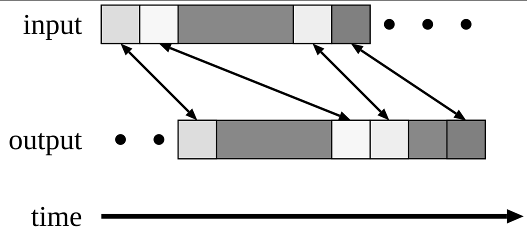


Figure 2.19: Transactions exit pipeline in-order

stream (I), an output stream (O) and two times (t_i and t_o) to match up corresponding input and output transactions. The relation is true if a transaction entered the pipeline at t_i and a transaction exited at t_o and the number of transactions that had entered the pipeline before t_i is the same that had exited before t_o. Using the relation *matchInOrder* to match up corresponding input and output transactions, we write the proof goal for our pipelined implementation as shown in Equation 2.6.

$$\begin{aligned}&\vdash \forall\ I,\ O : int\ \ signal\ .\\ &\quad diffAdd_P\ I\ O \implies\\ &\qquad \forall\ t_i,\ t_o\ .\ matchInOrder\ I\ O\ t_i\ t_o \implies diffAdd_S\ (I\ t_i)\ (O\ t_o)\end{aligned} \tag{2.6}$$

Equation 2.6 suffers from the deficiency that it may be vacuously true. The problem is that if transactions do not terminate from the pipeline (Figure 2.20), then *match* $I\ O\ t_i\ t_o$ will always be false. This will cause the antecedent of the implication to be false, and so the entire expression will be true. Additionally if transactions exit the pipeline that did not enter it (Figure 2.21), then we will be unable to match up corresponding transactions.

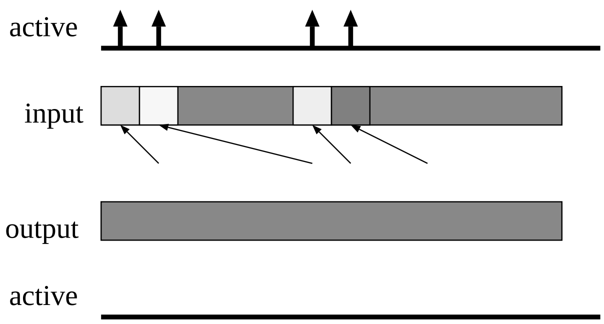


Figure 2.20: Transactions enter but don't exit

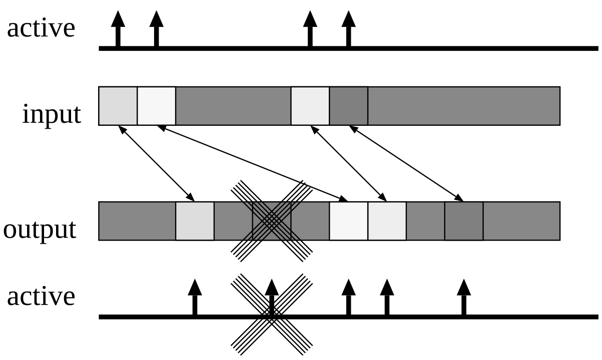


Figure 2.21: Transaction exits but didn't enter

To correct this deficiency we require that *matchInOrder* be one-to-one over the input and output streams. This stronger goal is shown in Equation 2.7. This goal has two new clauses, the first says that every input transaction corresponds to exactly one output transaction and the second says that every output transaction corresponds to exactly one input transaction.

$$
\begin{array}{l}
\vdash \forall\ I,\ O : int\ \ signal\ . \\
\quad diffAdd_P\ I\ O\ \Longrightarrow \\
\quad\quad \forall\ t_i\ .\ \exists!t_o\ .\ matchInOrder\ I\ O\ t_i\ t_o\ \& \\
\quad\quad \forall\ t_o\ .\ \exists!t_i\ .\ matchInOrder\ I\ O\ t_i\ t_o\ \& \\
\quad\quad \forall\ t_i\ t_o\ .\ matchInOrder\ I\ O\ t_i\ t_o\ \Longrightarrow \\
\quad\quad\quad diffAdd_S\ (I\ t_i)\ (O\ t_o)
\end{array} \tag{2.7}
$$

Definition 2.1: *Pipeline satisfies a specification (Copy of Definition 7.5)*

$$
\begin{array}{l}
(Pipe,\ Match,\ Constraints) \sqsupset Spec \ \hat{=} \\
\quad \forall I,\ O\ .\ (Pipe\ I\ O)\ \&\ (Constraints\ I\ O)\ \&\ (envOk\ I\ O)\ \Longrightarrow \\
\quad\quad PipeliningOk\ Match\ I\ O\ \& \\
\quad\quad \forall t_i,\ t_o\ .\ Match\ I\ O\ t_i\ t_o\ \Longrightarrow \\
\quad\quad\quad Spec\ (getTransComb\ (I\ t_i))\ (getTransComb\ (O\ t_o))
\end{array}
$$

Looking at Equation 2.7, we see that the basic structure of the goal will be the same for any pipeline, matching relation, and specification. We define the relation *satisfies* (shown in Definition 2.1 and displayed as "$\sqsupset$") that captures this structure. Using the relation *satisfies*, we write the goal in Equation 2.7 as shown in Equation 2.8. This pipeline does not put any constraints on the environment and so the third parameter on the left hand side of *satisfies* is *nullConstraints*.

$$\vdash (diffAdd_P,\ matchInOrder,\ nullConstraints) \sqsupset diffAdd_S \tag{2.8}$$

For Equation 2.8 we need to prove that *matchInOrder* is one-to-one over the input and output streams and that the implementation implies the specification. We temporarily delay describing how to verify that *matchInOrder* is one-to-one, and describe how to further simplify the verification of the functionality of the pipeline.

If we were to begin proving Equation 2.8, we would unfold the definition of *satisfies* and have two subgoals, one to show that *matchInOrder* is one-to-one and one to shown that the $diffAdd_P$ and *matchInOrder* imply $diffAdd_S$. The proof goal in Equation 2.9 corresponds to Equation 2.6 and the second subgoal.

$$
\begin{array}{l}
\vdash \forall\ I,\ O\ . \\
\quad diffAdd_P\ I\ O\ \Longrightarrow \\
\quad\quad \forall\ t_i,\ t_o\ .\ matchInOrder\ I\ O\ t_i\ t_o\ \Longrightarrow\ diffAdd_S\ (I\ t_i)\ (O\ t_o)
\end{array} \tag{2.9}
$$

Equation 2.9 is substantially more complex than our original goal for verifying the combinational logic implementation of the circuit (Equation 2.2). But, the datapaths of the two implementations (Figures 2.22 and 2.23) are very similar. We want to find a way to exploit this similarity to simplify the proof of Equation 2.9.

The strategy is to separate the verification of a pipeline into two independent tasks: verifying that the pipelining logic directs transactions through the pipeline correctly (termination

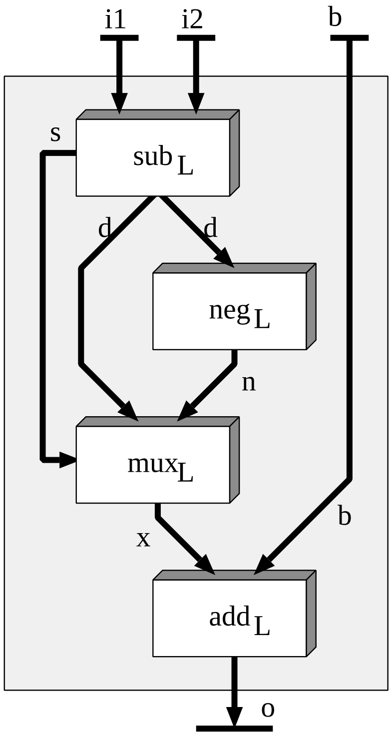


Figure 2.22: Combinational logic implementation

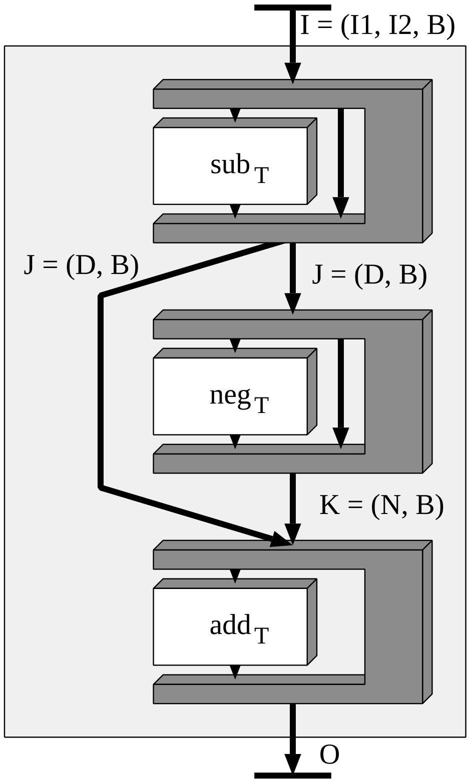


Figure 2.23: Pipelined implementation

according to the matching relation and freedom from interference) and verifying that the datapath and control logic compute the correct results. By separating the verification of the pipelining logic from the verification of the datapath, we can reason about a combinational representation of the datapath that does not contain any effects of pipelining. We do this by exploiting the fact that *satisfies* is transitive, as shown in Theorem 2.1.

Theorem 2.1: *Transitivity of satisfies (Copy of Theorem 7.2)*

$$
\begin{array}{l}
\vdash \forall PipeRecord,\ Comb,\ Spec\ .\\
\qquad PipeRecord \sqsupset Comb\ \&\\
\qquad (\forall T_i,\ T_o\ .\ Comb\ T_i\ T_o \implies Spec\ T_i\ T_o) \implies\\
\qquad\quad PipeRecord \sqsupset Spec
\end{array}
$$

This theorem says that if we can prove that a pipeline (*Pipe*), matching relation (*Match*), and constrains (*Constraints*) *satisfies* a combinational representation (*Comb*), then proving that *Comb* implies a specification (*Spec*) is sufficient to prove that the implementation and matching relation *satisfy Spec*. This theorem is proved by unfolding the definition of *satisfies* and applying Modus Ponens.

We create a combinational representation of a pipeline by replacing the datapath for each stage with its combinational representation and calculating the set of possible paths through the pipeline based on the control logic in the stages. The combinational representation of *diffAdd* ($diffAdd_M$ is shown in Definition 2.2. We represent each stage (sub_P, neg_P, add_P) with its combinational representation (sub_M, neg_M, add_M, plus the variables b_S and b_n to route values of B through sub_P and neg_P). The control logic in sub_P that directs transactions to either neg_P or add_P is modeled by the *if−then−else* after *sub*.

Definition 2.2: *Combinational representation of diffAdd (Copy of Definition 5.8)*

$$
\begin{array}{l}
diffAdd_M\ (i_1,\ i_2,\ b)\ o\ \hat{=}\\
\quad \exists s,\ d,\ b_s,\ n,\ b_n\ .\\
\qquad sub_M\ (i_1,\ i_2,\ b)\ (s,\ d,\ b_s)\ \&\\
\qquad if\ s\\
\qquad then\ \ neg_M\ (d,\ b_s)\ (n,\ b_n)\ \&\\
\qquad\qquad\ \ add_M\ (n,\ b_n)\ o\\
\qquad else\ \ add_M\ (d,\ b_s)\ o
\end{array}
$$

Once we have proved that the implementation and matching relation *satisfy* $diffAdd_M$, we apply Theorem 2.1 to Equation 2.8 and obtain Equation 2.10. This is in contrast to Equation 2.9, which results from unfolding the definition of *satisfies* in Equation 2.8 and verifying that *matchInOrder* is one-to-one.

$$\vdash \forall\ i,\ o.\ diffAdd_M\ i\ o \implies diffAdd_S\ i\ o \tag{2.10}$$

Both the left and right hand sides of Equation 2.10 are relations over transactions, while in Equation 2.9 $diffAdd_P$ is a relation over signals of transactions and *matchInOrder* is used to find corresponding transactions. In Equation 2.10 we have eliminated all of the affects of pipelining: contention for resources, time, and matching input and output transactions. A combinational representation of a pipeline contains a combinational representation of the datapath for each stage and the control logic to compute the paths that transactions follow through the pipeline. To verify that a pipeline and matching relation *satisfy* a combinational representation, we first prove that each stage *satisfies* its the combinational representation. Then we show that the paths through the combinational representation of the pipeline correctly represent the control and pipelining logic in the pipeline. This second condition requires showing that the control and pipelining logic guarantee that transactions will not interfere with each other, will follow the correct paths, and will terminate according to the matching relation. To simplify this proof, we have defined four specifications (*protocol*, *arbitration*, *control* and *ordering*) and have proved that if a pipeline, matching relation and combinational representation meet these specifications, then the pipeline and matching relation *satisfy* the combinational representation.

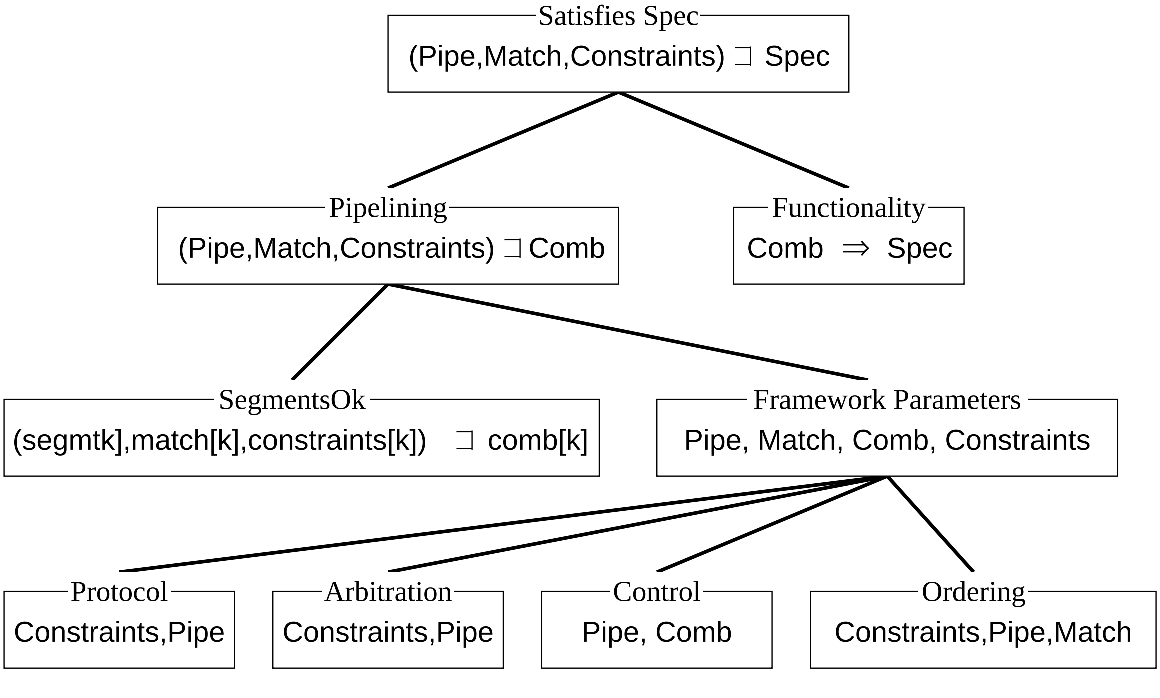


Figure 2.24: Hierarchy of proof obligations for verifying a pipeline

This process is summarized in Figure 2.24, which shows how the framework breaks down the task of verifying a pipeline. At the top is the relation *satisfies* (Definition 2.1), which is used to show that a pipeline and matching relation *satisfy* a specification. We introduce a combinational representation of the pipeline and split the proof into two obligations: showing that the pipeline satisfies the combinational representation and showing that the combinational representation implies the specification.

To show that the pipeline *satisfies* the combinational representation we prove that each stage *satisfies* its combinational representation and that the pipeline, combinational representation, and matching relation meet the specifications for the four parameters to the framework.

Theorem 2.2: *Pipelines that meet framework specifications satisfy their combinational representations (Copy of Theorem 7.3)*

$$
\begin{array}{l}
\vdash \forall\ \mathit{Pipe},\ \mathit{Match},\ \mathit{Constraints},\ \mathit{Comb}\ .\\
\quad (\mathit{CoresOk}\ \mathit{Pipe})\ \&\\
\quad (\mathit{ConstraintsOk}\ \mathit{Pipe}\ \mathit{Constraints})\ \&\\
\quad (\mathit{CombOk}\ \mathit{Pipe}\ \mathit{Comb})\ \&\\
\quad (\mathit{FworkParamsOk}\ (\mathit{Pipe},\ \mathit{Match},\ \mathit{Constraints}))\ \Longrightarrow\\
\qquad (\mathit{Pipe},\ \mathit{Match},\ \mathit{Constraints}) \sqsupset \mathit{Comb}
\end{array}
$$

Theorem 2.2 says that if a pipeline, its combinational logic representation, and the matching relation meet the specifications for the four framework parameters, then the pipeline and its matching relation *satisfy* the combinational representation. We also require that the core (datapath) for each stage is correct (*CoresOk*), the constraints of the segments are met (*ConstraintsOk*), and combinational representation of the pipeline reflects that paths through the pipeline (*CombOk*); but as described in Chapter 7, these obligations are easily solved when using the framework.

The result of proving Theorem 2.2 is that we know that $\mathit{diffAdd}_M$ is correct for $\mathit{diffAdd}_P$ and *matchInOrder*. When we include $\mathit{diffAdd}_P$ as a stage in the next higher level of the hierarchy, we can use this result to solve the first clause of the theorem for this stage. This allows us to naturally work our way through the structural hierarchy of the pipeline. The representation for each level is constructed by composing the representations for the stages at that level with the control logic. The base level of the hierarchy contains stages whose datapaths are purely combinational, so their datapath is the same as their representation.

Theorem 2.2 is the crux of the framework. It allows the verification of a pipeline to be separated into a number of largely independent tasks that, in almost all cases, are relatively easy to prove.

2.5 Introduction to Framework

This section briefly summarizes the terminology introduced in the previous sections and then introduces the four parameters to the framework. A pipeline is composed of a number of stages, which may themselves be pipelines or individual stages. We refer to an item of data, possibly paired with an instruction, as a *transaction*. Transactions flow through the stages in a pipeline. Each stage that a transaction passes through performs an *operation* on the data and passes the transaction onto another stage. Control information may come from two sources:

data-stationary control Control information is passed along with the data in the transaction.

data-dependent control Information calculated based on the data within the transaction.

The operation that a stage performs is determined by data-stationary control information. The next stage that a transaction is sent to may be determined by data-stationary and/or data-dependent control.

In most pipelines, when a stage wishes to transfer a transaction to another stage, the second stage may not be able to immediately receive the transaction. When this happens, the transaction is said to *collide*. These pipelines are said to contain *structural hazards*. Structural hazards can lead to problems with termination of transactions (deadlock and livelock) and interference between transactions.

For verification purposes we must be able to match up input and output transactions. In simple pipelines all transactions have the same *latency* (transit time through the pipeline), or exit the pipeline in the same order that they enter. In more complex pipelines, transactions may exit out-of-order. We may put constraints on an environment to prevent collisions and/or to ensure that exiting transactions can be uniquely paired with input transactions.

We can divide the process described above into four components:

1. transferring transactions between stages in the pipeline
2. selecting which stage is allowed to transfer a transaction to a given stage
3. performing operations on data and sending transactions on to next stages
4. matching up input and output transactions

These four components form the basis of the four parameters to the framework:

Protocol The segments of the pipeline must support some mechanism for communicating when one stage wishes to transfer a transaction to another stage and when a stage is able to receive an incoming transaction from another stage. Once two stages agree to a transfer, the appropriate transaction must be transferred from the first stage to the second. If the desired stage is unable to receive a transaction, the sending stage must hold onto the transaction and try to send again in the next clock cycle.

Arbitration When there are multiple stages which wish to transfer a transaction to the same next stage, the receiving stage must choose one of the stages in such a way as to ensure that all stages are eventually allowed to transfer their transactions.

Control When a transaction enters a stage, the stage must know which operation it is to perform; and when a transaction is ready to leave, the stage must know where to send it. This must be done in such a way that all paths through the pipeline lead to an exit.

Ordering For verification purposes, when a transaction exits the pipeline, we must be able to determine when the transaction entered the pipeline.

2.6 Conclusion

This chapter introduced our view of pipelines and a variety of classes of pipelines with different topologies and behavioral characteristics. We created a general definition of correctness for pipelines that guarantees that transactions terminate from pipelines and contain the correct data. Structural hazards appear in high-performance pipelines and lead to implementations that have out-of-order termination and/or large variances in latencies. This lead us to parameterize our definition of correctness by a matching relation for the pipeline. We also parameterized the definition by a set of constraints that the pipeline may impose on the environment.

We have proved a simple theorem that separates the verification of a pipeline into a combinational proof about the datapath and a data-independent proof about the pipelining circuitry. A second, and much more difficult theorem, allows us to use a framework of four parameters to simplify the verification of the pipelining circuitry into a number of separate and relatively simple tasks.

The informal specifications of the four parameters in Section 2.5 correspond to the the formal discussion of the parameters in Chapter 7. Using the specifications for the four framework parameters simplifies the process of verifying a pipeline, because the proof is broken up into several largely independent tasks. To further simplify this task, we have found a variety of schemes used in real pipelines that are commonly used to instantiate the parameters. These schemes are sufficient to characterize the instruction, integer, and floating-point pipelines in these microprocessors. The list includes the Digital Alpha 21064 [McL93], HP PA-Risc 7100 [AAD+93], IBM RS/6000 [Mis90], IBM PowerPC 601 [B+93], Intel Pentium [AA93], MIPS R4000 [MWV92], and Motorola 88110 [DA92].We also provide decision trees to assist designers in determining which standard instantiations are applicable for their pipeline.

Chapter 3

Description of Framework

3.1 Introduction

This section describes the four parameters to the framework (protocol, arbitration, control and ordering). For each parameter we give a general specification of the parameter that instantiations of the parameter must satisfy. One of the key features of the framework is that the parameters can be implemented using mixtures of hardware and constraints on the environment. The specifications for the parameters describe necessary *functionality*; only as much hardware as required to correctly produce the functionality required has to be used. There are degenerate cases in which parameters can be trivially implemented. For example, in a uniform-unifunctional pipeline each stage can perform only one operation and can send a transaction to only one other stage. So there is no hardware required to implement to control scheme.

We also provide a number of instantiations for each of the framework parameters. Each instantiation is used in one or more real microprocessors that our fictitious example microprocessor, the Adirondack in Chapter 6, is based on. Taken together, the instantiations are sufficient to characterize the instruction, integer, and floating-point pipelines in many microprocessors. As design styles evolve, new instantiations for the framework parameters will be found.

This chapter concludes with decision trees that guide the process of choosing instantiations for each parameter and verifying that a pipeline satisfies the corresponding specifications. Just as new instantiations can be added to the framework, they can also be added to the decision trees so as to incorporate them into the supported design and verification methodology.

3.2 Parameters to the Framework

This section describes the four parameters to the framework:

Protocol schemes describe how transactions are transferred between stages in the pipeline

Arbitration schemes specify how to prevent and/or handle structural hazards in the pipeline

Control schemes determine how transactions are routed through the pipeline and how stages know what operation to perform

Ordering schemes describe a method for matching up transactions as they leave the pipeline with transactions that entered the pipeline.

For each parameter we give a brief description, informally describe the specification, and list several instantiations. This information is summarized in a figure that contains the general specification and the specifications for the instantiations.

3.2.1 Protocol

The purpose of a protocol scheme is to move transactions from one stage to the next in the pipeline. In order to transfer a transaction between stages, the stage that is to receive the transaction needs to know that the other stage wants to send it a transaction. Conversely, the sending stage needs to know if the receiving stage is able to receive the transaction. We have named these two properties, the desire to send and the ability to receive, *request* and *accept*.

A protocol scheme defines the two virtual signals (*request* and *accept*) between each pair of stages that will send and receive transactions. The specification says that in a given cycle, a transfer must occur between two stages if the first stage sends a request to the second stage and the second stage sends an accept to the first. The specification also says that once a stage begins to request another stage, the request must be maintained until it has been accepted.

For some pipelines we can guarantee that every request will be immediately accepted. These pipelines are said to be free of *structural hazards*. If a pipeline is not free from structural hazards, then we need to prevent transactions from *colliding*. There are two sources of collisions: more than one transaction may wish to transfer into the same stage, or a transaction may wish to transfer into a stage that is already busy with another transaction.

An example of a degenerate protocol scheme is a uniform pipeline whose environment always accepts requests immediately. This class of pipeline does not contain any structural hazards. Each stage is connected to only one input stage, each transaction spends exactly the same amount of time in each stage, and the environment accepts requests immediately. A stage requests the next stage in the pipeline whenever it has a transaction and requests are always accepted. The only hardware that is required for this protocol scheme is some way of denoting that a transaction is leaving the pipeline. This is commonly done using an active signal that travels through the pipeline with a transaction or a special NOP opcode in pipelines with data-stationary control.

We have found two commonly used protocol schemes. The first, which we refer to as a *transit* scheme, ensures that whenever a transaction enters a pipeline, it is guaranteed not to collide with any transaction already in the pipeline — it can "transit" through the pipeline. In the second, more general method, transactions are allowed to proceed through the pipeline until just before a collision. When a transaction cannot proceed any further without colliding, it *stalls* until the potential collision is cleared.

Anytime a designer can verify that a potential collision will not happen, no hardware needs to be built to prevent that collision. Many times designers put constraints on the environment, such as requiring the environment to always accept transactions immediately or requiring certain properties about the input stream of transactions, that greatly simplify the protocol hardware inside the pipeline.

The MIPS R3000 microprocessor uses a transit protocol scheme. The compiler is required to schedule code so that instructions are not issued to the pipeline if they might conflict with an instruction already executing. We use this scheme in the MIPS R3000 version of the ADK instruction pipeline (Section 6.3). An alternative implementation of a transit protocol scheme is to use a single circuit at the entry of the pipeline that only accepts transactions if it can guarantee that they will not collide in the pipeline. This scheme appears in the ADK floating-point pipeline (Section 6.5). The DEC Alpha 21064 based instruction pipelines for the ADK (Section 6.4) use general protocol schemes, as does a variation on the NVAX floating-point pipeline (Section 6.5).

General

- When two stages agree to transfer a transaction, the transaction must be transferred.
- If a request is not accepted, it must be maintained until it is accepted.

Degenerate

- Every stage receives transactions from only one source
- Every transaction spends the same amount of time in each stage
- Environment accepts requests immediately

Transit

- When a transaction is issued to a pipeline it will not collide with any transaction already in the pipeline.

Figure 3.1: Specifications for protocol schemes

3.2.2 Arbitration

The purpose of an arbitration scheme is to prevent deadlock and livelock within the pipeline. An arbitration scheme ensures that whenever a stage wants to send a transaction to another stage, the receiving stage will eventually agree to receive the transaction. The specification for arbitration schemes says that if all of the requests from a pipeline to the environment are eventually accepted, then all of the internal stages must eventually accept all incoming requests. An arbitration scheme is responsible for choosing which requesting stage has the highest priority.

In a degenerate arbitration scheme each stage receives input transactions from only one stage. For these pipelines there is no need for arbitration hardware, because there is a never a choice as to which stage the next transaction will come from. Our example pipeline for this scheme is the MIPS R3000 style single-issue instruction pipeline in Section 6.3.

Sometimes, even though a stage is connected to multiple input stages, we can show that only one request will be active at a time. We refer to this as an *exclusive* arbitration scheme. The implementation only needs to select whichever request is active. We know that we can use an exclusive arbitration scheme for pipelines with a transit protocol scheme, because a transit protocol scheme guarantees that transactions accepted into the pipeline will not encounter any collisions.

For pipelines that allow multiple simultaneous requests to the same stage, we need to provide a method for prioritizing the requests. In many pipelines we can assign *static* priorities to requests based upon the stage that the request comes from. We have found a set of topological constraints and rules for assigning priorities to stages that is sufficient for a static arbitration scheme to be used. These rules do not restrict the input stream of transactions or the behavior of the stages. The ADK floating point pipeline (Section 6.5) can use either an exclusive or static arbitration scheme, depending on whether we use a transit protocol scheme or not.

In the topological rules for a static arbitration scheme, we flatten the structural hierarchy of a pipeline down to stages that can only hold one transaction each (recall that in general stages may themselves contain pipelines). This flattened structure defines a graph where each stage is a node and two nodes are connected by a directed edge if a transaction can transfer from the first stage to the second. The rules refer to *cycles* and *skips*. A cycle is a list of stages connected by edges where each stage has an edge pointing to the next stage in the list and the last stage has an edge pointing to the first stage in the list. A skip is an edge that connects two stages that are also connected via a list of two or more edges. Two skips overlap if the destination stage of the first skip is after the source stage for the second skip.

There are five rules in our specification:

1. Each stage is a member of only one cycle
2. Each potential cycle has exactly one entry stage
3. Stages in cycles have higher priority than other stages
4. No overlapping skips
5. Non-skipping transactions have higher priority

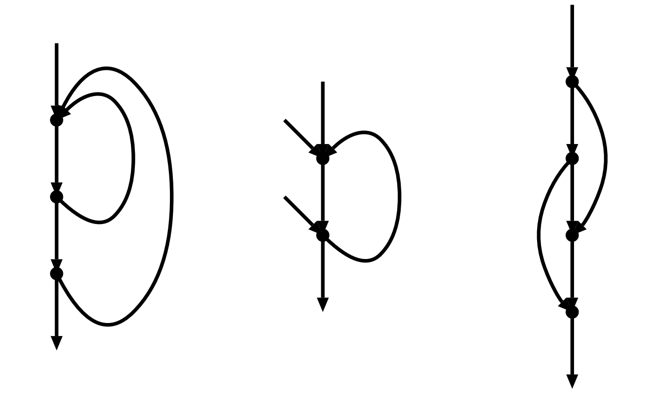


Figure 3.2: Graphs of non-compliant pipelines

The non-compliant topologies are illustrated in Figure 3.2.
Another specification for the static arbitration scheme allows non-compliant topologies, but puts behavioral restrictions on the pipeline. We call these the "no-passing" rules for a static arbitration scheme. If we assign static priorities to stages so that whenever a transaction begins to request a stage, we can guarantee that no transaction from later in the input stream can request the desired stage from a higher priority stage until the first transaction has been accepted, then we can use a static arbitration scheme. The DEC Alpha 21064 style instruction pipeline (Section 6.4) uses a static arbitration scheme that satisfies the no-passing rules but not the topological rules.

3.2.3 Control

A control scheme determines how a stage decides what operation to perform on each transaction and where to send the transaction when it is done in the stage. The specification for control schemes says that every possible path through the pipeline must eventually exit the pipeline. Another way to view this is to say that a single transaction traveling through an empty pipeline will eventually exit the pipeline. The purpose is to disallow pipelines in which transactions are meant to loop through a sequence of stages an infinite number of times. If transactions were allowed to do this, then even if the pipeline was implemented correctly, transactions would not exit the pipeline.
A stage can decide what operation it is to perform on a transaction based on *data-stationary* and/or *time-stationary* control information. Data-stationary control information is carried along with the transaction as it proceeds between stages. Time-stationary information is provided to the stage at each clock cycle, independent of which transaction is in the stage. A stage can determine where to send a transaction when it is done based on the data-stationary information with which the transaction entered the stage, data-dependent

General

- All requests must eventually become the highest priority request to the stage they are requesting
- All highest priority requests must eventually be accepted.

Degenerate

- Each stage receives input transactions from only one stage

Exclusive

- Each stage is guaranteed to have at most one request active at a time

Static prioritization without cycles

- There are no potential cycles of requests in the pipeline.
- No overlapping skips
- Non-skipping transactions have higher priority

Static prioritization with cycles

- No overlapping skips
- Non-skipping transactions have higher priority
- Each stage is a member of at most one potential cycle of requests
- Each potential cycle of requests has exactly one entry stage
- Stages within cycles have higher priority than other stages

Figure 3.3: Specifications for arbitration schemes

information generated within the stage (such as the result of a comparison or an overflow condition), and/or the current value of time-stationary control signals.
Simple instruction pipelines (*e.g.* the MIPS R3000 style instruction pipeline in Section 6.3) are unifunctional and so do not require any control information. Most complex pipelines in microprocessors use data-stationary control to determine what operation to perform on the data and to route transactions through the pipeline (*e.g.* the DEC Alpha 21064 style instruction pipeline in Section 6.4). The NVAX floating-point pipeline (Section 6.5) uses both data-stationary and data-dependent control.
If the topology of the pipeline is acyclic, then we trivially know that every path through the pipeline leads to an exit. If the topology does have cycles, we need to verify that none of the loops can be repeated an infinite number of times. For simple loops it may be easiest to use a state-exploration method to check that every possible path through the pipeline will terminate. This technique works for the floating-point pipeline for the Adirondack (Section 6.5) (and the IBM RS/6000 floating-point pipeline), where double-precision multiplies iterate through a loop exactly twice. In more complex loops, counters may be used to iterate through the loop. (*e.g.* the divide stage in the Adirondack floating-point pipeline, which is based on the VAX 8600 model 200).

General

- All possible paths through the pipeline eventually exit the pipeline.

Topological rules

- The topology of a pipeline is a directed acyclic graph.

Loops terminate

- Every path that contains a cycle (one or more stages used multiple times in the same path) is guaranteed to exit the loop(s).

Figure 3.4: Specifications for control schemes

3.2.4 Ordering

Ordering schemes are used when verifying a pipeline to check that the output transactions contain the correct results. To do this, we must be able to match up an output transaction with the input transaction that caused it and the clock cycle when the transaction was issued to the pipeline. 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. We need to know when the transaction was issued to the pipeline because the result could be dependent upon time-stationary control information that was sent to the pipeline while the transaction was traversing through the pipeline.

Each ordering scheme defines a relation that takes an index (c_i) into the input stream of transactions and an index (c_o) into the output stream and returns true if and only if the transaction exiting the pipeline at c_o resulted from the transaction entering the pipeline at c_i. In single issue pipelines (one input and one output port) the indices are times and in superscalar pipelines (multiple input and output ports) the indices are a time and port identifier.

The simplest way to match input and output transactions is if the pipeline has a fixed latency. In these pipelines all transactions will exit the pipeline a given number of cycles after they enter the pipeline. Our MIPS R3000 based instruction pipeline (Section 6.3) exhibits this behavior.

The next simplest case is pipelines with in-order termination. For these pipelines, examining the request and accept signals between the pipeline and the environment is sufficient to determine the correspondence between input and output transactions. Example pipelines using this scheme include a single issue instruction pipeline in which transactions could stall inside the pipeline.

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 transactions with the same tag have the same latency or exit the pipeline in-order. In the RS/6000 floating-point pipeline, transactions with the same opcode have the same latency. In the ADK floating-point pipeline transactions with the same opcode have different latencies, but exit in-order. In the DEC Alpha 21064 style instruction pipeline (Section 6.4), instructions with the same opcode exit in-order. Another method for assigning tags is to use the destination of the transaction. This method is also an alternative for the ADK (Section 6.5) and the IBM RS/6000 floating-point pipelines, because only one transaction with a given destination register can be in the pipeline at a time.

3.3 Commercial Microprocessors and the Framework

In this section we use the framework to characterize the instruction, integer, and floating-point pipelines for a number of commercial microprocessors. For each pipelines we describe how it instantiates the four framework parameters. The results are summarized in Table 3.1. The microprocessors are, in alphabetical order, the Digital Alpha 21064 [McL93], HP PA-Risc 7100 [AAD+93], IBM RS/6000 [Mis90], IBM PowerPC 601 [B+93], Intel Pentium [AA93], MIPS R4000 [MWV92], and Motorola 88110 [DA92]. These designs span the range from the CISC architecture of the Pentium to the ultra-RISC architecture of the Alpha 21064. All of these designs, except MIPS R4000 are superscalar: the Alpha 21064, HP 7100, and Intel Pentium can issue up to two transaction per clock cycle; the PowerPC 601 can issue up to three transactions per cycle; and the RS/6000 can issue up to four transactions per cycle.

In addition to instruction, integer, and floating-point pipelines, microprocessors require

General

- Define *Match* relation that takes two stream indices (c_i and c_o); returns true if the transaction exiting the pipeline at c_o resulted from the transaction issued at c_i

Uniform latency

- All transactions have the same latency through the pipeline.

In order

- Transactions exit the pipeline in the same order that they were issued

Tags: Uniform latency

- All transactions with same tag have the same latency through the pipeline.

Tags: In order

- Transactions with the same tag exit the pipeline in the same order that they were issued

Unique active tags

- Only one transaction with a given tag is in the pipeline at a time.

Figure 3.5: Specifications for ordering schemes

method for processing branches and an interface to memory. Branch units are either integrated into the instruction pipeline, as is done in the IBM RS/6000 and Intel Pentium, or as single stage execution units (*e.g.* Alpha 21064 and PowerPC 601). In the microprocessors surveyed here, branches and load/store transactions are either handled by the instruction pipeline, integer pipeline, or by independent units that are unpipelined or very simple pipelines. The interesting aspects of these transactions that are related to structural hazards are subsumed by our discussions of the instruction and/or integer pipelines, so we do not analyze branches or load/store transactions.

Table 3.1: Framework Instantiations for Several 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	●		●	●		●	●														

3.3.1 Instruction Pipelines

All of the pipelines except the MIPS R4000 are superscalar, which means that multiple transactions may be issued simultaneously. Instruction pipelines send output transactions to the execute units of the microprocessor (*e.g.* branch, integer, floating-point, or load/store). These units may not always be able to accept an incoming transaction; or data may not be available for a transaction when it wants to fetch its operands in the instruction pipeline. Either of these circumstances will cause transactions to stall in the instruction pipeline. This potential of stalling transactions within the pipeline forces the use of a general protocol scheme in all of the instruction pipelines surveyed here. One notable exception to this is the MIPS R3000, the precursor to the R4000, which is able to use a degenerate protocol scheme. This is because the compiler is responsible for scheduling code in such a way that collisions never occur.

Instruction pipelines perform essentially the same computation on all transactions. Because of this, in a single-issue instruction pipeline, all transactions follow the same path, and so the R4000 uses a degenerate arbitration scheme. Two of the superscalar microprocessors (HP 7100 and M88110) also use a degenerate arbitration scheme in their instruction pipelines. A superscalar instruction pipeline is constructed by using multiple single-issue instruction pipelines executing in parallel. The HP 7100 determines the exit port of a transaction before it enters the instruction pipeline. The exit port is determined by which execution unit the transaction will transfer to. The identity of the execution is determined by the opcode, which is predecoded in the instruction cache. Transactions are issued to the same side of the pipeline that they will exit from, so each stage is connected to only one previous stage. The M88110 can send a transaction from either exit port in the instruction pipeline to any execution unit. Here again, there is no need for a transaction to change sides of the pipeline and thus each stage has only one previous stage.

The IBM RS/6000 sends both output transactions from the instruction pipeline to both the integer and floating-point pipelines. (Up to four instructions are fetched at a time; branch and condition-code transactions are handled completely within the instruction pipeline.) Unlike the M88110, which can also send both outputs from the instruction pipeline to any execution pipeline, the RS/6000 does not use a degenerate arbitration scheme. The RS/6000 increases hardware utilization with a sophisticated "bubble squashing" algorithm.* Each stage in the RS/6000 is connected to two previous stages. The routing algorithm guarantees that each stage will never receive simultaneous requests from both of its two previous stages. This is an exclusive arbitration scheme.

The PowerPC 601 uses a concept somewhat similar to the RS/6000 in order to use an exclusive arbitration scheme in a superscalar pipeline, but the implementation is very different. The instruction pipeline in the PowerPC 601 is essentially an eight slot queue and some decode logic. The integer pipeline can receive transactions from the last slot in the queue. The floating-point pipeline and branch unit can receive transactions from any of the last four slots. At each clock cycle that the pipeline does not stall, each transaction in the queue is either shifted down or issued to an execution unit. The transaction in the last slot requests the integer pipeline, floating-point pipeline, or branch unit based upon its opcode. Each of the oldest (furthest down in the queue) floating-point and branch transactions requests its respective execution unit. The queue can shift each transaction up to three slots per clock cycle (a maximum of three transactions can be issued per cycle). Each transaction that does not request an execution unit requests the entry in the queue that it will be transferred to in the next cycle.

The remaining two microprocessors (the Alpha 21064 and Intel Pentium) use static arbitration schemes. In both pipelines the entry port of a transaction is determined by its location in memory and the exit port is determined by the execution unit that the transaction will be transferred to. This necessitates the ability to switch a transaction from one side of the

*Bubble squashing transfers transactions into vacant stages when a stall occurs instruction pipeline to the other. This is done by using one pair of stages as "swap" stages — transactions leaving a swap stage may request to swap to the other side of the pipeline. If a pair of simultaneously issued transactions both need to exit from the same side of the pipeline (*e.g.* two floating-point transactions) then both swap stages will request the same next stage. This contention is resolved by statically prioritizing the requests based on the stage. In both the Alpha 21064 and Intel Pentium, the left half of the pipeline receives transactions from one slot earlier in the instruction stream than the right. Giving the left swap stage priority over the right side preserves program order as transactions traverse the pipeline.

In an instruction pipeline transactions use each stage at most once, so all of the instruction pipelines use a no-loops control scheme.

The R4000 is a single-issue pipeline and so it uses an in-order ordering scheme. The HP 7100 is superscalar, but it also uses an in-order ordering scheme. It predecodes transactions, so the exit port of a transaction is determined solely by its entry port. To achieve an in-order scheme we sort transactions by their port and then by time. For example, given two ports, one named A and one B with $A < B$, we say that for any two times (t_a and t_b): $(A,t_a) < (B,t_b)$. This method of sorting transactions means that the HP 7100 can use an in-order ordering scheme without any need to tag transactions.

The five other instruction pipelines use a tags-in-order ordering scheme, where transactions are tagged with the execution unit that they will use. The execution unit is determined by the opcode of the transaction. For example, all integer transactions exit in the same order that they enter, but a floating-point transaction issued before an integer transaction may exit after the integer transaction.

3.3.2 Integer Pipelines

None of the integer pipelines, except the Pentium, allow a transaction to enter if it might collide with a transaction already in the pipeline. The Pentium, because of its need to support the extensive addressing modes of the x86 architecture, has much more complex integer pipelines (there are two in the Pentium) than the other microprocessors. The addressing modes result in several additional ALUs for address calculation and an increased difficulty in detecting dependencies between transactions. If two register/memory transactions are issued simultaneously, they are serialized inside the integer pipelines, because it is too difficult to detect a dependency between the transactions. This serialization causes subsequent transactions in the pipeline to stall for one clock cycle; hence the need for a general protocol scheme.

Because they do not allow transactions to enter if they will collide with another transaction, all of the integer pipelines other than the Pentium use transit protocol schemes. In the HP7100, Pentium, and M88110 integer pipelines, all transactions follow the same path, so these pipelines use a degenerate arbitration scheme. The M88110 has two identical integer ALUs and a bit-field unit. All three of these units are single stages. They have one clock cycle latencies and use degenerate arbitration schemes.

The RS/6000, MIPS R4000 and PowerPC 601 implement integer divide instructions by having divide transactions iterate through a loop around a stage. Except for the Alpha 21064, in the integer pipelines that do not support divide instructions, transactions use each stage at most once, and so the pipelines use a no-loops control scheme. The integer multiply instruction in the Alpha 21064 loops through the multiply stage just as divide transactions in the RS/6000, MIPS R4000 and PowerPC 601 loop through their divide stages. These pipelines use a "loops" control scheme. The Pentium supports integer divide operations, but the operation is handled as a sequence of microinstructions. This means that, to the integer pipelines, a divide instruction appears as series of independent transactions, where each transaction uses each stage at most once. Thus, the Pentium's integer pipelines use a no-loops control scheme.

In the HP 7100, Pentium, and M88110 integer pipelines there is one entry and one exit port and all transactions follow the same path. Transactions exit in the same order that they enter, thus the pipelines use an in-order ordering scheme. The Alpha 21064, RS/6000, MIPS R4000, and PowerPC 601 have paths that contain loops (the multiply in the Alpha 21064 and the divide for the others). In the Alpha 21064, RS/6000 and MIPS R4000, short latency transactions (*e.g.* additions) skip over the stages in the loops and so a multiply or divide may exit later than a short latency transaction that was issued after it. Thus, transactions may exit the pipeline out-of-order. Because of the need to observe data dependencies between transactions, only one transaction with a given destination register is active in the pipeline at a time. By tagging transactions by their destination register, we can use a tags-unique ordering scheme. The divide transactions in the PowerPC 601 loop around a stage in the main pipeline. Transactions can not pass other transactions and so the PowerPC 601 uses an in-order ordering scheme.

3.3.3 Floating-Point Pipelines

All of the floating-point pipelines use a transit protocol scheme: the pipeline will not accept an input transaction if it will collide with a transaction already in the pipeline. The HP 7100, Intel Pentium, and M88110 use degenerate arbitration schemes. The HP 7100 and M88110 have separate add, multiply and divide units (the HP 7100 has a divide/square root unit). This configuration allows each stage to have one previous stage. For the divide units though, there is a loop. Each divide unit has two sources: the input to the floating-point pipeline and the loop back from itself. The divide unit uses an exclusive arbitration scheme, but the loop is not visible to the overall pipeline and so the pipeline uses a degenerate scheme. In the Pentium floating-point pipeline, like its integer pipelines, the divide instruction (and other complex instructions) appear as a series of independent transactions, each of which follows the same path. This allows the Pentium floating-point pipeline to use a degenerate arbitration scheme.

Like the HP 7100 and M88110, the Alpha 21064 and MIPS R4000 have specialized divide stages, but divide transactions also utilize other stages in the pipeline. The R4000 has a second source of loops: square roots loop through the add stage in the pipeline. This results

in exclusive, rather than degenerate, arbitration schemes for the Alpha 21064 and MIPS R4000. The IBM RS/6000 and PowerPC 601 implement divides by looping through stages in the main body of the pipeline. These pipelines also use exclusive arbitration schemes.
All of the floating-point pipelines except the Pentium utilize loops to implement divides. The Pentium uses a no-loops control scheme and the others use a loops control scheme.
Those pipelines that have independent divide units or specialized divide stages allow non-divide transactions to pass divide transactions in the pipeline. These pipelines use tags-unique ordering schemes. The IBM RS/6000 and PowerPC 601 use the main body of the pipeline to do divides and so other transactions can not pass divides. These two pipelines use in-order ordering schemes. All floating-point transactions follow the same path through the Pentium's floating-point pipeline and so it too uses an in-order ordering scheme.

3.4 Conclusion

This chapter described the four parameters to the framework and commonly used instantiations for these parameters. To aid designers in choosing the right instantiations for a pipeline, we provide decision trees. When using a decision tree a designer is asked a number of questions about the pipeline and incurs correctness obligations that must be checked to ensure that the pipeline meets the specification for the chosen instantiation. Once the instantiations for the framework parameters have been chosen, the designer can proceed to build the pipelining circuitry to transform each datapath into a stage in the pipeline.
The framework defines virtual functional units whose behavior the pipeline stages implement. The virtual functional units prescribe required behavior, they do not constrain how the behavior is implemented. Simple pipelines can often implement this behavior with very little hardware, while more complex pipelines will require additional hardware.
We describe a method for selecting appropriate implementations for the functional unit based on the instantiations of the framework parameters. By following this method of using the decision trees and implementations the verification of a pipeline is carried out in an organized manner as the design progresses. As design styles evolve the framework can be extended with new choices in the decision trees and additional implementations.

Chapter 4

Using the Framework

4.1 Introduction

This chapter describes how the framework is used to design and verify pipelines. Section 4.2 presents a set of decision trees that step users through a series of questions leading to instantiations of the framework for a particular pipeline. The majority of the chapter (Section 4.3) outlines the process of designing, implementing and informally verifying a pipeline. The concepts presented in this chapter are demonstrated in Chapter 5 using a small pipeline that contains a structural hazard.

4.2 Decision Trees for Framework Parameters

This section describes the decision trees for the four framework parameters. The trees aid designers in choosing instantiations for the parameters for a pipeline and verifying that the pipeline meets the specifications for the parameters. Designers begin at the "START" node in the upper left of each decision tree. Most of the interior nodes of the tree (the rectangles) are questions that the designer must answer about the pipeline. The exceptions are obligations that must be checked before proceeding further in the tree; these are shown as rectangles with rounded ends. If a question can not be trivially answered, it is followed in the tree by an obligation to prove that the answer is correct. This allows designers to easily remember what they must prove to verify that their pipeline meets the specifications. The terminal nodes in the tree are specifications. A terminal may either be a standard instantiation or may be the general specification. The obligations incurred while traversing the path up to a standard instantiation are sufficient to informally show that the pipeline meets that specification. If a terminal node is the general specification, then some amount of additional reasoning must be done to verify that the specification is met. The legend for each figure describes the decisions, obligations, or additional specifications for each node. The decision trees are extensible: as additional standard instantiations are found, they may be incorporated into the decision trees.

4.2.1 Decision Tree for Protocol Schemes

The decision tree for the protocol scheme is shown in Figure 4.1; the legend appears in Table 4.1. This tree begins by asking if the environment to the pipeline can wait before accepting a request. If it can, then we use the general specification, because the first criteria for a degenerate or transit protocol scheme is that the environment accepts requests immediately.

Table 4.1: Decisions for checking protocol schemes

Exit env can delay requests	The environment does not have to immediately accept requests to exit the pipeline.
General	Use the specification for a general protocol scheme.
Env issues every cycle	The environment issues a new transaction every cycle.
Linear topology	Each stage receives inputs from only one other stage and sends outputs to only one other stage.
All stages have same latency	All stages have the same latency for all transactions.
Degenerate	The pipeline meets the specification for a degenerate protocol scheme.
Accept a req if not busy	If a stage is not busy and has at least one request, then it must accept a request and receives the accepted transaction.
Tree topology	Different paths through the pipeline never converge on same stage.
Env prevents collisions	The environment never requests the pipeline if immediately accepting the request could lead to that transaction colliding with a transaction already in the pipeline or issued in the same cycle (in the case of multiple entry stages)
Transit	The pipeline meets the specification for a transit protocol scheme.
Entry prevents collisions	The entry stages to the pipeline never accept a request if it could lead to that transaction colliding with a transaction already in the pipeline or issued in the same cycle (in the case of multiple entry stages).

We then check if the environment must issue a new transaction every cycle. If the pipeline receives a new transaction every cycle, we have a linear topology, and all stages have the same latency then we can use a degenerate protocol scheme.

If the environment is not constrained to issue a transaction every cycle, then we must show that whenever a stage is not busy and has at least one active request, it accepts a request.

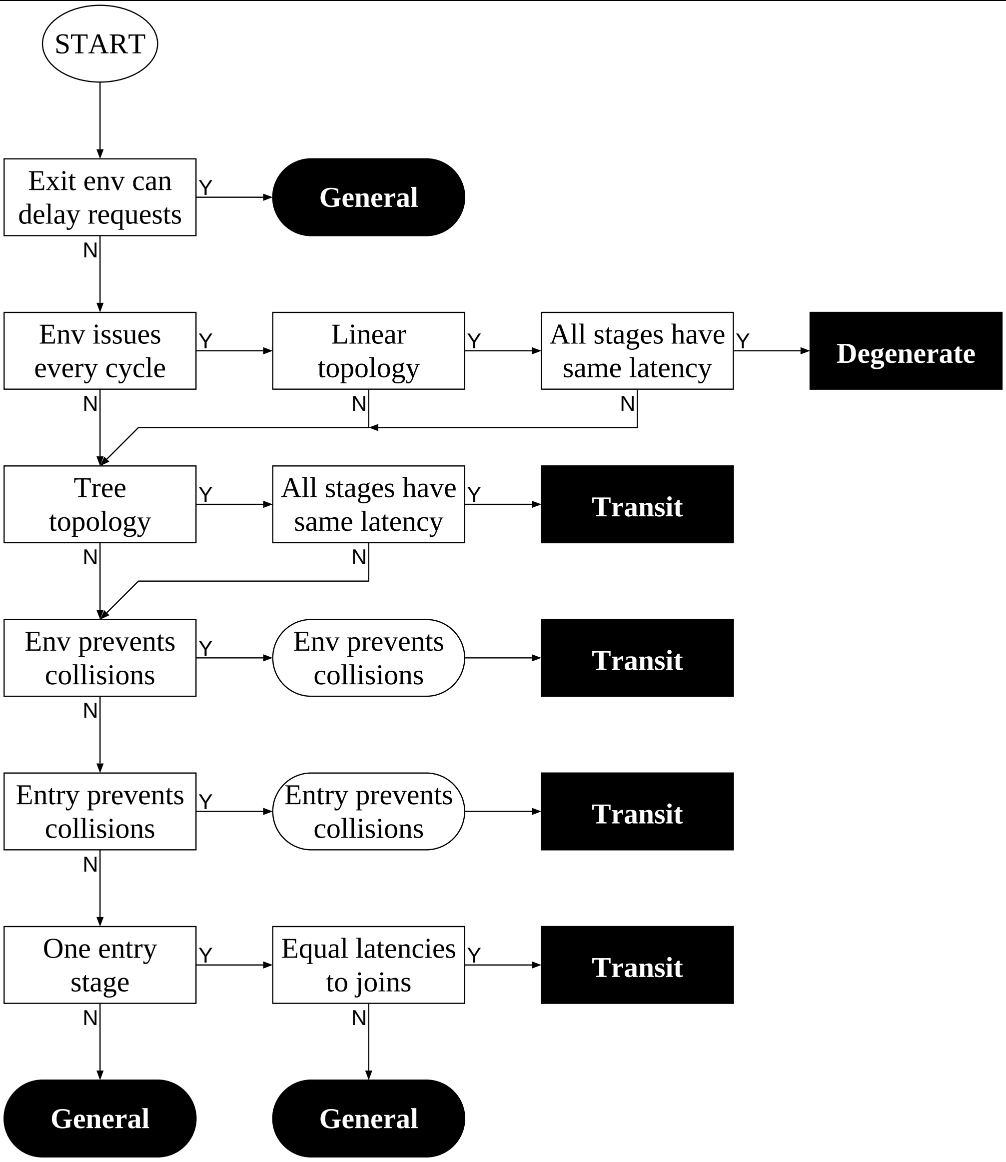


Figure 4.1: Decision tree for checking protocol schemes

We then check if the topology of the pipeline is a tree. If the pipeline is a tree, then we know that each stage can receive input transactions from only one source stage. This means that there is no chance of having collisions caused by two or more transactions trying to transfer into the same stage at the same time.

We next check if we can have collisions caused by a transaction attempting to transfer into a stage that is busy. If all stages have the same latency, the environment accepts requests immediately, and each stage has only one potential source, then we have met the degenerate protocol scheme. If all stages do not have the same latency, then, unless we include further restrictions, it is possible that a transaction may attempt to transfer into a stage that is still busy processing the previous transaction.

We return now to the node that checked if the topology of the pipeline is a tree. If the pipeline is not a tree, then we have multiple paths through the pipeline that lead to the same stage. One way to prevent collisions at this node of the decision tree is to constrain the environment to not request the pipeline if accepting such a request could lead to a collision with a transaction that is already in the pipeline or is being issued in the same clock cycle. If this constraint is met, then we have conformed to the requirements of a transit protocol scheme. Another way to meet the transit protocol specification is by requiring that the entry stage(s) of the pipeline does not accept requests that could lead to the transaction colliding with a transaction that is either already in the pipeline or is being issued at the same time.

4.2.2 Decision Tree for Arbitration Schemes

The decision tree (Figure 4.2 and Table 4.2) for arbitration schemes begins by checking if the topology of the pipeline is a tree. If it is, then we know that each stage will receive input transactions from only one other stage. Thus we do not need any arbitration logic and we have a degenerate arbitration scheme.

The second question asks if the pipeline uses a transit protocol scheme. If it does, then we know that there will be at most one simultaneous request to any stage. If there is at most one request at a time, then we only need to make sure that the arbitration scheme selects the request that is active. This meets the specification for an exclusive arbitration scheme.

Next, we check if the topology of the pipeline is a directed acyclic graph (DAG). If it is, then we know that we can never have a cycle of stages, where each stage in the cycle is requesting to use the next stage in the cycle. If the pipeline is a DAG and it meets the topological rules for acyclic pipelines, then we can use the static prioritization scheme to decide which request is the highest priority.

If the pipeline does not obey the topological rules for a static arbitration scheme, it may meet the no-passing rules for a static arbitration scheme. Checking the no-passing rules is more involved than checking the topological rules and may require some verification effort. Because of this, we add an obligation to verify that the rules are met. If we do not use a static arbitration scheme, we might want to use the control logic in each stage to select the next stage to transfer to such that we never have two stages requesting the same next stage.

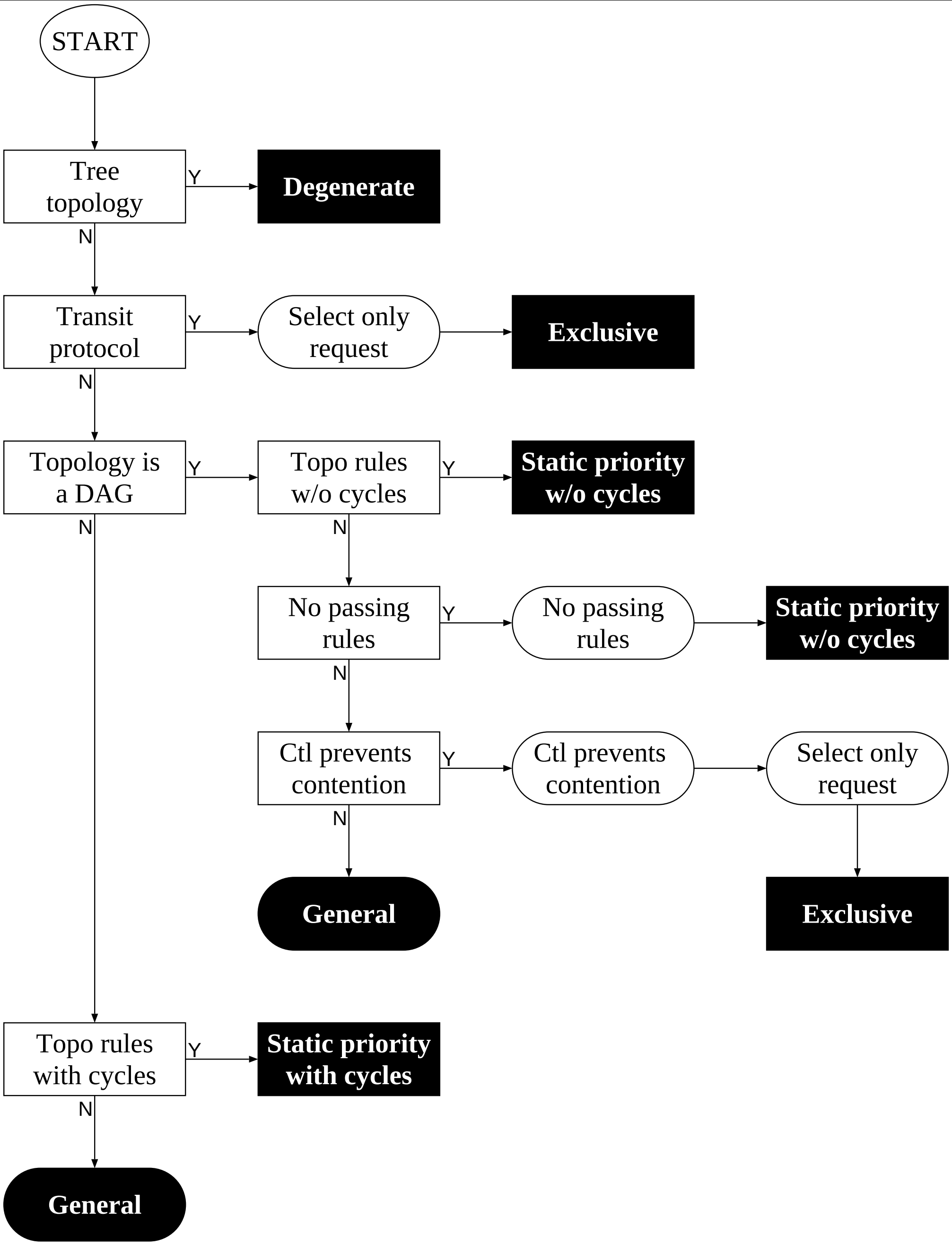


Figure 4.2: Decision tree for checking arbitration schemes

Table 4.2: Decisions for checking arbitration schemes

Tree topology	Different paths through the pipeline never converge on same stage.
Degenerate	The pipeline meets the specification for a degenerate arbitration scheme.
Transit protocol	The protocol scheme meets the specification for a transit protocol.
Select only request	Arbitration scheme must always select the one stage with the incoming request.
Exclusive	The pipeline meets the specification for an exclusive arbitration scheme.
Topology is a DAG	The topology of the pipeline is a directed acyclic graph.
Topo rules w/o cycles	The pipeline does not contain any reconvergent branches or overlapping skips; non-skipping transactions have priority over skipping transactions (Section 3.2.2.
Static priority w/o cycles	The pipeline meets the specification for a static arbitration scheme with an acyclic.
No passing rules	The no-passing rules for a static arbitration scheme are met.
Control prevents contention	The control logic for each stage ensures that it never requests the same next stage as another stage.
General	The pipeline must be shown to meet the specification for a general arbitration scheme.
Topo rules with cycles	In addition to the rules for pipelines without cycles: each stage is a member of at most one potential cycle of transactions, each cycle has one entry point, and stages within cycles have higher priority.
Static priority with cycles	The pipeline meets the specification for a static arbitration scheme with a cyclic topology.

If we use this prioritization scheme, we have a pipeline where each stage is guaranteed to have at most one active request at a time. To ensure that requests are eventually accepted, we require that the arbitration logic select the request which is active. If we do not use the control logic of the stage to route transactions so that collisions are prevented, we can use some other scheme and verify it against the general arbitration specification.
We return now to the node in the decision tree that asked if the pipeline was a directed acyclic graph. If the pipeline has cycles and meets the topological rules for pipelines with cycles, then we can use the static prioritization scheme for cyclic pipelines. Otherwise, we use the general specification.

4.2.3 Decision Tree for Control Schemes

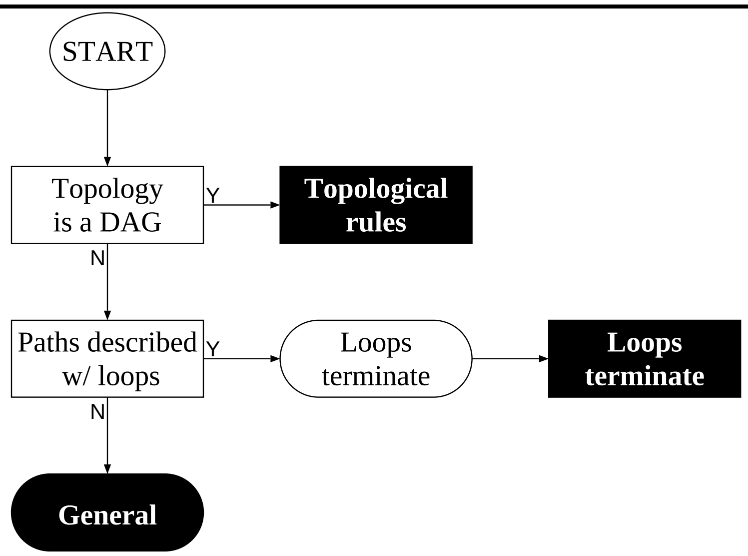


Figure 4.3: Decision tree for checking control schemes

The specification of a control scheme is that all possible paths through the pipeline terminate. Verifying that the control logic leads transactions to follow the correct path is part of verifying the functionality of the pipeline. The decision tree (Figure 4.3 and Table 4.3) for control schemes begins by checking if the topology of the pipeline is acyclic. If it is, then all paths will lead to an exit and so we have met the topological rules for a control scheme. If the topology contains cycles, then we have to show that all loops in the pipeline terminate.

4.2.4 Decision Tree for Ordering Schemes

We begin the decision tree for verifying ordering schemes (Figure 4.4 and Table 4.4) by checking if there is only one entry stage to the pipeline. We next check if transactions exit

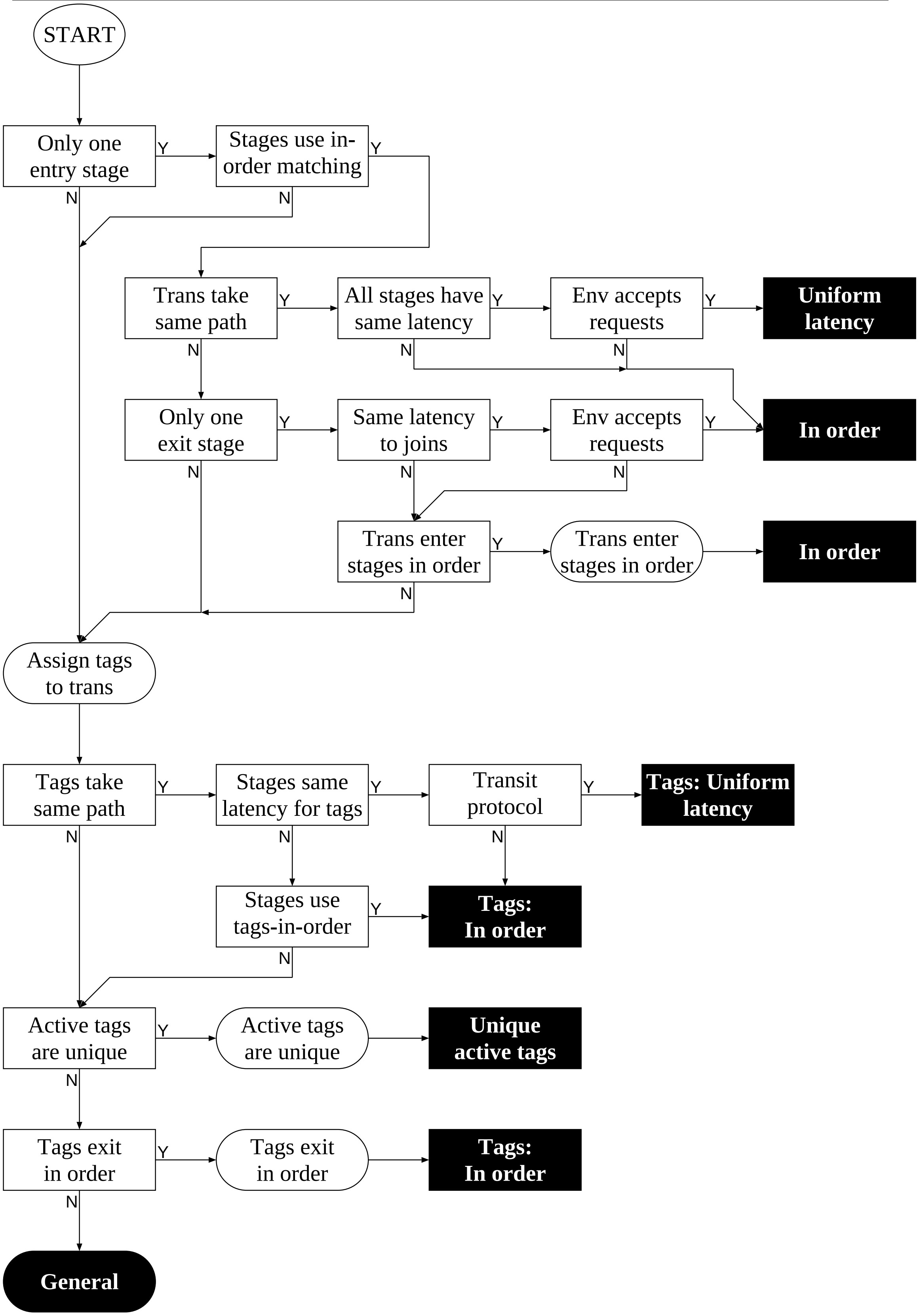


Figure 4.4: Decision tree for checking specification for ordering

Table 4.3: Decisions for checking control scheme

Topology is a DAG	The topology of the pipeline is a directed acyclic graph.
Topological rules	The specification for a control scheme using topological rules is met.
Paths described with loops	The paths through the pipeline that reuse stages can be described as looping through a set of stages.
Loops terminate	To meet the looping control scheme, every loop in the pipeline must be shown to terminate.
General	The general control scheme must be shown to hold.

each stage in the same order that they enter it; this is true if all of the stages in the pipeline use an in-order ordering scheme. These two properties can greatly simplify the ordering scheme. If both of these properties are met, then we check if all transactions take the same path. If they do, and if all stages have the same latency and the environment accepts requests immediately, then all transactions have the same latency through the pipeline.
Even if all stages don't have the same latency, or the environment does not always accept requests immediately, we know that transactions will exit the pipeline in the same order that they entered the pipeline. Also, if transactions don't all take the same path through the pipeline; but there is only one exit stage, all paths that lead to the same stage have the same latency, and the environment immediately accepts request, then we know that transactions will exit the pipeline in-order. Finally, if we can prove that transactions enter each stage in the pipeline in the order that the entered the pipeline, we know that transactions will exit the pipeline in-order.
If all transactions with the same tag take the same path through the pipeline, then we might be able to use an ordering scheme where transactions with the same tag have the same latency or exit the pipeline in-order. In order for transactions with the same tag to have the same latency through the pipeline, each stage must have the same latency for all transactions with the same tag and the pipeline must be free of stalls. A pipeline is guaranteed free of stalls if we were able to use a transit protocol scheme.
If each stage has the same latency for all transactions with the same tag, but the pipeline is not free of stalls, then transactions with the same tag will exit the pipeline in-order. If not all stages have the same latency for all transactions with the same tag, but transactions with the same tag exit the stages in-order (stages have a tags-in-order ordering scheme with the same tags as the pipeline), then we know that transactions with the same tag exit the pipeline in-order.
If transactions with the same tag don't exit stages in-order, or transactions with the same tag don't take the same path through the pipeline, then we might be able to assign tags to transactions so that there is only one transaction with a given tag inside the pipeline at a time. If we can prove that at any point in time the active tags are unique, then we use the unique-tags ordering scheme.

Table 4.4: Decisions for checking ordering schemes

Only one entry stage	There is only entry stage to the pipeline.
Stages use in-order matching	All of the stages use an in-order ordering scheme.
Trans take same path	All transactions follow the same path through the pipeline
All stages have same latency	All stages have the same latency for all transactions.
Env accepts requests	The environment immediately accepts requests.
Uniform latency	The uniform latency scheme is valid.
Only one exit stage	There is only one exit stage in the pipeline.
Same latency to joins	Paths that lead to the same stage have the same latency.
In order	The in-order ordering scheme is valid.
Trans enter stages in-order	Transactions enter each stage in the same order that they entered the pipeline.
Assign tags to trans	Choose algorithm to assign tags to transactions.
Tags take same path	All transactions with the same tag take the same path through the pipeline.
Stages same latency for tags	All transactions with the same tag (using same tags as in pipeline) have the same latency through a given stage.
Transit protocol	The pipeline uses a transit protocol scheme.
Tags: Uniform latency	An ordering scheme where transactions with equal tags have the same latency is valid.
Stages use tags-in-order	All stages use a tags-in-order ordering scheme with the same tags that the pipeline uses.
Tags: In order	An ordering scheme where transactions with equal tags exit the pipeline in-order is valid.
Active tags are unique	At any point in time there is only one transaction with a given tag in the pipeline.
Unique active tags	A unique-active-tags ordering scheme is valid.
Tags exit in-order	Tags exit the pipeline in the same order they enter the pipeline.
General	The specification for a general ordering scheme must be shown to hold.

If more than one transaction with the same tag can be inside the pipeline simultaneously, but transactions with the same tag don't all follow the same path, we may still be able to prove that transactions with the same tag exit the pipeline in-order. If all else fails, we can fall back on the general specification for the ordering scheme.

4.3 Design and Verification Methodology

4.3.1 Introduction

Even for relatively simple pipelines with structural hazards, designing a correct and efficient implementation on the first attempt is very unlikely when using an ad-hoc design process. Even in seemingly simple designs, structural hazards introduce potential sources of errors that are not immediately apparent. By organizing the design process and specifying correctness conditions at each step, the framework helps guide the user to an optimized and correct implementation.

This section describes how the framework is used to design and verify pipelines. Chapters 5 and 6 contain example pipelines and demonstrate how the framework is used. The purpose of this section is to give a flavor for the design process and to act as a central reference for general information that is used in the examples. For this reason, some of the more detailed information contained here may not be clear until reading about the example pipelines.

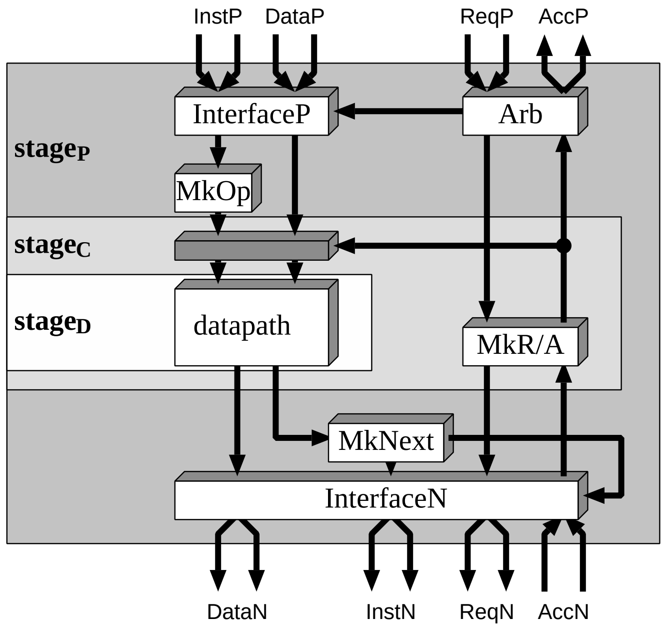


Figure 4.5: A stage is built by adding pipelining logic to the datapath

The framework specifies virtual functional units and signals that each stage must implement. As with all parts of the framework, these virtual functional units and signals describe required functionality; only as much hardware as needed to produce the required functionality must be used. The framework aids the user in implementing these virtual functional units and signals by providing decision-trees and implementations of hardware components. The decision trees step the user through the process of finding instantiations for the four framework parameters (protocol, arbitration, control, and ordering). Each instantiation of a framework parameter has an associated implementation for the virtual functional units and signals.

Figure 4.5 shows a canonical stage with its virtual functional units. At the center is the datapath ($stage_D$), then the core ($stage_C$) and then the pipeline stage ($stage_P$). The datapath ($stage_D$), core ($stage_C$), and stage ($stage_P$) are analogous to the combinational-logic implementation ($stage_L$), sequential implementation ($stage_T$) and stage ($stage_P$) used when we gave an *ad-hoc* description of designing a pipeline in Section 2.4.

In this section we describe the steps required to design and verify a pipeline using the framework. The steps may be done in a variety of different orders; we describe one possible order.

1. Define the specification for the pipeline ($pipe_S$)
2. Build an implementation and combinational representation for the datapath for each stage ($stage_D$, $stage_M$)
3. Find the instantiations of the framework parameters for the pipeline
4. For each stage:
   (a) Find the class of each stage
   (b) Choose a matching relation
   (c) Create a core by adding registers and request/accept generation circuitry as needed to the datapath ($stage_C$)
   (d) Create a pipeline stage by adding interface, arbitration, and control circuitry to the core ($stage_P$)
   (e) Build the combinational representation of the entire stage from the control circuitry and the combinational representation of the datapath
   (f) Prove that the combinational representation of each stage is correct wrt to the implementation
5. Define request/accept virtual signals between stages
6. Create the combinational representation ($pipe_M$)
7. Choose the matching relation
8. Prove that the pipeline conforms to the framework parameters
9. Prove that the combinational representation implies the specification

4.3.2 Datapath Design

In a top-down design style, engineers begin the design of a pipeline with a specification of the desired behavior. They then find an algorithm to implement this behavior and partition the computation into tasks for the datapaths of the stages. A great deal of research has investigated the process of finding an algorithm that can be efficiently implemented on a pipeline and partitioning the algorithm into tasks for the stages. The framework concentrates on the subsequent steps in the design process.

For any stage that can contain at most one transaction and whose computation can always be done in one clock cycle, we construct the combinational representation by replacing each sequential hardware component in the datapath with the equivalent combinational component. For example, if the sequential inverter in Equation 4.1 is in a datapath, we replace it with the combinational inverter in Equation 4.2.

$$inv_T\ i\ o \implies \forall t\ .\ o\ t = \neg(i\ t) \tag{4.1}$$

$$inv_L\ i\ o \implies o = \neg i \tag{4.2}$$

For more complicated stages, we need to define a combinational representation and verify that it correctly models the stage. This requires choosing a matching relation for the stage, which is described in Section 4.3.6.

4.3.3 Choosing Instantiations for the Framework Parameters

There are four parameters to the framework: protocol, arbitration, control and ordering. Each parameter has a decision tree (Section 4.2) that guides the designer in finding instantiations for the parameters. Once we have found instantiations for the framework parameters, we design the pipelining circuitry for the stages.

4.3.4 Classes of Stages

The core of a stage is built around the datapath. The core stores the transaction while the computation is being performed and while the stage is stalled. It also generates request and accept signals as needed. These tasks are all part of the protocol scheme. To assist in designing the protocol circuitry, we have classified stages into four categories:

Degenerate	A new transaction enters each cycle and the environment always accepts requests immediately.
Unit-Delay	New transactions do *not* have to enter each cycle, but the environment has to immediately accept requests.
General	New transactions do *not* have to enter each cycle, and the environment does *not* have to accept requests immediately.
Zero-delay	The stage contains purely combinational hardware (almost always a very small amount); transactions spend zero clock cycles in the stage.
Multi-delay	Transactions may need to spend more than one cycle within the datapath.

The first three classes of stages correspond to degenerate, transit, and general protocol schemes where each transactions need to use the datapath for exactly one clock cycle. In a general protocol scheme transactions may stall and be forced to remain in a stage for more clock-cycles, but they are ready to leave after one cycle. In some pipelines though, we need a datapath that contains a small amount of glue hardware. If this hardware is sufficiently fast,* we can implement the hardware without adding an extra clock cycle to the latency of the pipeline. We model this as a stage in which transactions spend zero clock cycles. Uses of this class of stages include connecting multiple stages to the same input or output port of a pipeline.

In some pipelines we need a datapath that can contain at most one transaction, but where the transaction may use the datapath for more than one clock cycle. In the other classes of stages, we know when to request the next stage, but for this class this is dependent upon the datapath. So, when a transaction arrives, we alert the datapath and then wait for the datapath to inform us that the transaction is ready to exit.

There are three different protocol schemes: degenerate, transit, and general. For a pipeline with a degenerate protocol scheme, all stages are degenerate. No hardware is needed for request or accept signals, and the storage hardware is just a register that latches in a new transaction on every clock cycle. For a transit protocol scheme, we know that stages cannot stall, so all transactions are guaranteed to spend exactly one cycle within each stage. All of the stages in these pipelines are zero-delay or unit-delay stages. In a general protocol scheme, the default class for each stage is "general" (except for those requiring extra time in the datapath, which are "multi-delay" stages). Sometimes though, we can prove that some of the stages will never stall. We classify these stages as "unit-delay" and use the simpler circuitry and matching relation.

4.3.5 Generating Request and Accept Signals

We have developed a set of implementations for the request/accept signals, their generating logic, the storage hardware for transactions, and the matching relation for each of the five

*Or we are successful in using retiming techniques to rearrange the combinational logic and registers.

classes of stages. (Tables 4.5 and 4.6). Throughout this thesis we suffix signals with "N" and "P" for "next" and "previous". So, "ReqP" is the request signal from the previous stage. In Table 4.5, a "*true*" in a column for a request or accept signal means that it is always true and a "$\surd$" means that it needs to be implemented in hardware. If an entry is the name of another signal, then the two signals are equivalent. Stages that will not stall latch in new data on each clock cycle and propagate their request signal through using a simple register. Stages that may stall require a register that selectively loads in new data (Equation 4.4) and extra hardware to generate request and accepts (Equations 4.5 or 4.6). The matching relations from Table 4.5 are shown in Definitions 4.1 – 4.3.

Table 4.5: Standard implementations for stage parameters

	ReqP	AccP	ReqN	AccN	R/A Logic
Zero-delay	$\surd$	*AccN*	*ReqP*	$\surd$	—
Degenerate	*true*	*true*	*true*	*true*	—
Unit-delay	$\surd$	*true*	$\surd$	*true*	reg_T
General	$\surd$	$\surd$	$\surd$	$\surd$	*mkReqAcc*
Multi-delay	$\surd$	$\surd$	$\surd$	$\surd$	*mkReqAccDel*

Table 4.6: Standard implementations for stage parameters (storage and matching)

	Storage	Match
Zero-delay	—	*matchDelay* 0
Degenerate	reg_T	*matchDegen*
Unit-delay	reg_T	*matchDelay* 1
General	$regLd_T$	*matchUnique*
Multi-delay	$regLd_T$	*matchUnique*

$$reg_T\ I\ O \implies \forall t.\ O\ (t{+}1) = I\ t \tag{4.3}$$

$$regLd_T\ Ld\ I\ O \implies \forall t.\ O\ (t{+}1) = \mathit{if}\ Ld\ \mathit{then}\ (I\ t)\ \mathit{else}\ (O\ t) \tag{4.4}$$

In Equation 4.5, we request the next stage (*ReqN*) if we received a transaction in the previous cycle or we were stalled in the previous cycle. We are able to accept an incoming transaction (*AccP*) if we were able to send out our transaction or were not processing a transaction in this cycle.

$$\begin{aligned}
&mkReqAcc\ (reqP,\ accP)\ (reqN,\ accN) \implies \\
&\forall t\ . \\
&\quad reqN(t{+}1) = (reqP\ t)\ \vee\ ((reqN\ t)\ \&\ \neg(accN\ t))\ \& \\
&\quad accP(t{+}1) = (accN\ (t{+}1))\ \vee\ \neg(reqP\ t)
\end{aligned} \tag{4.5}$$

In Equation 4.6, we add two extra signals (*iReqP* and *iReqN*) for internal request-previous and request-next. These signals are an input to, and an output from the datapath. We also add an internal signal (*busy*) that remembers if there is a transaction inside the datapath that has not exited yet.

$$
\begin{array}{l}
\mathit{mkReqAccDel}\ (\mathit{reqP},\ \mathit{accP})\ (\mathit{iReqP},\ \mathit{iReqN})\ (\mathit{reqN},\ \mathit{accN}) \implies \\
\ \exists \mathit{busy}\ . \\
\quad \forall t\ . \\
\qquad \begin{array}{ll}
\mathit{iReqN}(t{+}1) & = \mathit{iReqP}\ t\ \& \\
\mathit{reqN}(t{+}1) & = (\mathit{iReqN}\ (t{+}1))\ \vee\ ((\mathit{reqN}\ t)\ \&\ \neg(\mathit{accN}\ t))\ \& \\
\mathit{busy}(t{+}1) & = (\mathit{ReqP}\ (t{+}1))\ \vee\ ((\mathit{busy}\ t)\ \wedge\ \neg(\mathit{iReqN}\ (t{+}1)))\ \& \\
\mathit{accP}(t{+}1) & = (\mathit{accN}\ (t{+}1))\ \vee\ \neg(\mathit{busy}\ t)
\end{array}
\end{array}
\tag{4.6}
$$

When using a transit protocol scheme where the entry stage is responsible for preventing collisions, we require additional hardware to maintain state information about the pipeline. A standard implementation is Davidson's shift-register based circuit [DSTP75]. This circuit and a method for designing it for individual pipelines have been described extensively in literature [Kog81, Sto90]. The circuit is a state machine that keeps track of the internal state of the pipeline in a set of shift registers, one for each path through the pipeline. When an incoming transaction requests the pipeline, the output bit of the appropriate shift-register shows whether the path that the transaction will take will encounter any collisions. If no collision will occur, then the transaction is accepted and the internal state updated to reflect the path of the new transaction.

4.3.6 Matching Relations

The matching relations are independent of the data flowing through the stage (the variables I and O in Definitions 4.1–4.3) and (except for zero-delay stage) have the same request and accept signals as the stages for which they are applicable. "Degenerate" stages do not have either request or accept signals, "unit-delay" stages have requests but no accept signals, and "general" stages have both requests and accepts. A zero-delay stage simply connects its request and accept signals straight through: *ReqP* to *ReqN* and *AccN to AccP*. It does not store a transaction and can be used in between two stages of any class. If a virtual signals is implemented as a signal in hardware, we use a wire to connect the stages. When a request or accept signal is always true, we optimize away the wire.

Definition 4.1: *Degenerate matching*

$$\mathit{matchDegen}\ I\ O\ t_i\ t_o \ \hat{=}\ t_o = t_i{+}1$$

4.3.7 Connecting Stages Together

The framework defines five virtual functional units that all stages must implement (Figure 4.6). The InterfaceP and InterfaceN functional units are part of the protocol scheme,

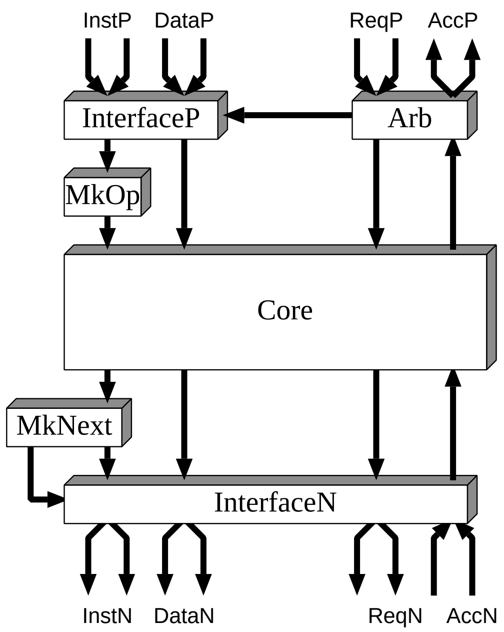
Definition 4.2: *Matching with uniform latency*

$$matchDelay\ n\ (I,\ reqP)\ \ (O,\ reqN)\ t_i\ t_o\ \ \hat{=}\ \ (reqP\ t_i)\ \ \&\ \ t_o = t_1 + n$$

Definition 4.3: *Matching with unique active transactions*

$$\begin{aligned} &matchUnique\ (I,\ (reqP,\ accP))\ \ (O\ (reqN,\ accN))\ \ \hat{=} \\ &\qquad (reqP\ t_i)\ \ \&\ \ (accP\ t_i)\ \ \& \\ &\qquad (t_o = nextTimeTrue\ (\lambda t\ .\ reqN\ t\ \&\ accN\ t)\ (t_i{+}1)) \end{aligned}$$

the Arb functional unit is for the arbitration scheme, and the MkOp and MkNext units are for the control scheme. The implementations of the Interface and Arb units are given in Tables 4.7 and 4.8. The two control units, MkOp and MkNext particular to an individual datapath and pipeline, so we do not provide standard implementations for them.



InterfaceP	Interface the stage to its source (previous) stages
Arb	Arbitrate between multiple requests
MkOp	Determine what operation the datapath is to perform
MkNext	Determine the next stage to go to and calculate data-stationary control information
InterfaceN	Interface the stage with its destination (next) stages

Figure 4.6: Canonical stage with sub-components

In a degenerate arbitration scheme each stage is only connected to one source and one destination stage. For these pipelines, there is no need for an arbitration unit and the interface units are simply wires connecting the internal signals of the stage with the signals of the source and destinations stages. When there is more than one source or destination stage, we use multiplexers and de-multiplexers for the interface units. We also need an arbitration unit to select which source stage has the highest priority. For Exclusive and Static-priority arbitration schemes, a priority encoder is the standard implementation of the arbitration unit. For these two implementations to work together correctly, we need to assign an index number to each source stage and ensure that Arb and InterfaceP use the same index for the same stage. Similarly, MkNext generates the index for InterfaceN, so we must ensure that these two units use same set of indices.

Table 4.7: Standard Implementations of Interface Units

Protocol Scheme	InterfaceP	InterfaceN
Degenerate	—	—
Transit	mux/demux	mux/demux
General	mux/demux	mux/demux

Table 4.8: Standard Implementations of Arbitration Unit

Arbitration Scheme	Arb
Degenerate	—
Exclusive	priority encoder
Static-priority	priority encoder

4.3.8 Combinational Representation and Matching Relation for the Pipeline

The combinational representation of a pipeline is built by composing the combinational representations of the stages with the control circuitry. The combinational representation describes the paths through the pipelines. It is built by symbolically executing the control circuitry and combinational representations of the cores of the stages. Section 7.3.1 describes in detail how to calculate the combinational representation of a pipeline and Appendix B.1 demonstrates this technique with the *diffAdd* pipeline introduced in Chapter 2 and described in detail in Chapter 5.

The ordering scheme for a pipeline determines the matching relation to be used and the method for calculating tags, if one is needed. Table 4.9 lists the ordering schemes and their matching relations. The matching relation for a uniform latency pipeline is *matchDelay* (Definition 4.2), which is also used for unit-delay and zero-delay stages. The other matching relations in Table 4.9are shown in Definitions 4.4 – 4.7.

Table 4.9: Ordering schemes and matching relations

	Relation	Definition
Uniform-latency	*matchDelay*	(4.2)
In-order	*matchInOrder*	(4.4)
Tags: Uniform-latency	*tagsDelay*	(4.5)
Tags: In-order	*tagsInOrder*	(4.6)
Tags: Unique	*tagsUnique*	(4.7)

Definition 4.4: *Matching function for in-order ordering scheme*

$$
\begin{array}{l}
matchInOrder\ (I,\ (reqP,\ accP))\ (O,\ (reqN,\ accN))\ t_i\ t_o\ \ \hat{=} \\
\quad let \\
\quad\quad activeI\ t\ \hat{=}\ (reqP\ t)\ \ \&\ \ (accP\ t) \\
\quad\quad activeO\ t\ \hat{=}\ (reqN\ t)\ \ \&\ \ (accN\ t) \\
\quad in \\
\quad\quad (activeI\ t_i)\ \ \&\ \ (activeO\ t_o)\ \ \& \\
\quad\quad (count\ activeI\ t_i = count\ activeO\ t_o) \\
\quad end
\end{array}
$$

Definition 4.5: *Matching relation for tags-uniform-latency ordering scheme*

$$
\begin{array}{l}
tagsDelay\ n\ (I,\ activeI)\ (O,\ activeO)\ c_i\ c_o\ \hat{=} \\
\quad (activeI\ c_i)\ \ \&\ \ (activeO\ c_o)\ \ \& \\
\quad (I\ c_i = O\ c_o)\ \ \&\ \ (c_o = c_i + n)
\end{array}
$$

Definition 4.6: *Matching relation for tags-in-order ordering scheme*

$$
\begin{array}{l}
tagsInOrder\ (I,\ activeI)\ (O,\ activeO)\ t_i\ t_o\ \hat{=} \\
\quad let\ tag = I\ t_i\ in \\
\quad\quad (O\ t_o = tag)\ \ \&\ \ (activeI\ t_i)\ \ \&\ \ (activeO\ t_o)\ \& \\
\quad\quad (count\ (\lambda t\ .\ (I\ t = tag)\ \&\ activeI\ t)\ t_i = \\
\quad\quad\quad count\ (\lambda t\ .\ (O\ t = tag)\ \&\ activeO\ t)\ t_o) \\
\quad end
\end{array}
$$

Definition 4.7: *Matching relation for tags-unique ordering scheme*

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

4.3.9 Verification

When using the framework to prove that a pipeline and matching relation *satisfy* a specification, there are two main goals that need to be solved:

1. The pipeline meets the specifications for the four parameters to the framework
2. The combinational representation of the pipeline implies the specification

There are three other minor goals, but two of them are taken care of automatically when working with with the framework and the third is almost always quite easy to prove. These minor goals are:

1. The individual stages *satisfy* their combinational representations
2. The constraints that the stages impose on the pipeline are met
3. The combinational representation of the pipeline accurately represents the implementation

If the pipeline was built using the steps described in this section and the standard implementations were used for the virtual functional units and matching relation, then the first minor obligation is very easy to fulfill. Alternatively, if some of the stages are themselves pipelines and they were built and verified using the framework, then this fulfills the first obligation. Most constraints that stages impose on pipelines affect only one other stage and are direct consequences of that stages behavior. Informally generating the combinational representation of a pipeline is very easy and it is easy to check that the combinational representation is accurate.

For the two significant obligations, we must show that the instantiations of the framework parameters are correct and that the combinational representation implies the specification. When using the decision trees to choose instantiations for the framework parameters, we answer questions about the pipeline and environment and incur obligations. To prove that the pipeline conforms to the specifications for the framework parameters, we prove the obligations that were incurred while traversing the trees.

The second goal is to prove that the combinational representation of the pipeline implies the specification. This shows that the datapath of the pipeline is correct. This goal is free from all aspects of pipelining; normal verification techniques can be used.

4.3.10 Multiple-port Pipelines

The discussion in this section so far has described pipelines with one input port and one output port. The framework can be used with multiple-port pipelines, such as instruction pipelines for super-scalar microprocessors. The implementation of a pipeline with two input and two output ports is a relation over two signals whose elements are pairs of transactions. When verifying the functionality of such a pipeline the framework allows us to abstract away from the fact that two transactions may enter the pipeline simultaneously. Verifying the functionality of a single-issue instruction pipeline with n classes of instructions would require n different cases, while if we had to verify all possible pairs of transactions, we would need n^2 cases.

In some multi-port pipelines, the behavior of the pipeline depends upon the port that a transaction enters or exits. To capture this, we make the combinational representation and specification relations over pairs containing a port and a transaction. This allows the combinational representation and specification to reason about a single transaction and still capture correctness properties associated with having multiple ports.

Equation 4.7 shows the canonical functionality goal for a pipeline with one input and one output port. Equations 4.7 shows the types for the relations, streams, and stream indices.

$$pipe\ I\ O\ \ \&\ \ match\ I\ O\ t_i\ t_o\ \ \Longrightarrow\ \ spec\ (I\ t_i)\ (O\ t_o) \tag{4.7}$$

$$pipe\ \ : ((time{\rightarrow}trans) \rightarrow (time{\rightarrow}trans))\ relation \tag{4.8}$$
$$match\ \ : ((time{\rightarrow}trans) \rightarrow (time{\rightarrow}trans) \rightarrow time \rightarrow time)\ relation \tag{4.9}$$
$$spec\ \ : (trans{\rightarrow}trans)\ relation \tag{4.10}$$

$$I,O\ \ : time \rightarrow trans \tag{4.11}$$
$$t_i,t_o\ \ : time \tag{4.12}$$

To make the types of the implementation, matching relation, combinational representation, and specification for multiport pipelines all match, we introduce the function *squeeze* (displayed as "$\downarrow$"), which maps a stream of tuples of transactions (one for each port) that are indexed by time into a stream of individual transactions that are indexed by a time and a port. This gives us the canonical functionality goal shown in Equation 4.13.

$$pipe\ I\ O\ \ \&\ \ match\ ({\downarrow}I)\ ({\downarrow}O)\ tp_i\ tp_o\ \ \Longrightarrow\ \ spec\ (({\downarrow}I)\ tp_i)\ (({\downarrow}O)\ tp_o) \tag{4.13}$$

For pipelines with two input and two output ports the types of the relations, streams, and variables are as shown in Equations 4.14 – 4.18. Equation 4.19 shows the definition of *squeeze* ($\downarrow$) for pipelines of this type.

$$
\begin{array}{lll}
pipe & : ((time{\rightarrow}(trans{\times}trans)) \rightarrow (time{\rightarrow}(trans{\times}trans)))\ relation & (4.14)\\
match & : ((time{\times}port \rightarrow trans) \rightarrow (time{\times}port \rightarrow trans) & \\
 & \quad\quad \rightarrow time{\times}port \rightarrow time{\times}port)\ relation & (4.15)\\
spec & : (port{\times}trans \rightarrow port{\times}trans)\ relation & (4.16)\\
 & & \\
I,O & : time \rightarrow trans{\times}trans & (4.17)\\
tp_i,tp_o & : time{\times}trans & (4.18)\\
\downarrow & \mathrel{\hat{=}} \lambda(t,p)\ (T_0,\ T_1)\ .\ (p,\ if\ p{=}0\ then\ T_0\ else\ T_1) & (4.19)
\end{array}
$$

The only operations that we perform on indices into streams is to test for equality and increment. These operations are performed in the matching relations and in a few of the lemmas used in the verification of the framework. With single-port pipelines, equality and increment have their usual meaning over the integers. With multi-port pipelines, an index contains the identifier of a port and a time. Equality requires that both the port and time are identical and increment simply increments the time component.

4.4 Conclusion

In this chapter we covered the use of the framework to specify, design, and informally verify pipelines. In Chapter 5 we illustrate the use of these techniques with a simple pipeline and in Chapter 6 we give an overview of the use of the framework with the instruction and floating point pipeline of a superscalar microprocessor.

Chapter 5

DiffAdd: The Design and Verification of a Small Pipeline

5.1 Introduction to DiffAdd Example Pipeline

In this chapter we step through the process of designing and verifying the simple pipeline, *diffAdd*, which was introduced in Section 2.4. Chapter 6 contains an overview of how the framework can be used with larger pipelines, namely a super-scalar pipelined RISC microprocessor. Due to the complexity of that example, we do not describe it in detail as much detail as *diffAdd*.

The *diffAdd* example calculates $|I_1 - I_2| + B$. We begin with a combinational implementation of the circuit. We model combinational logic as having zero delay, and so there is no notion of time. We then create a non-pipelined sequential implementation of *diffAdd*. The purpose of including this example is to contrast the verification of a non-pipelined circuit with that of a pipelined circuit. The bulk of the chapter demonstrates how the framework is used to design and verify the pipelined implementation of *diffAdd*.

5.2 Non-pipelined Implementations of DiffAdd

In this section we briefly describe the specification, design, and verification of two non-pipelined implementations of *diffAdd*. The first is a purely combinational implementation and the second is sequential.

5.2.1 Combinational Logic Implementation

Definition 5.1 shows the specification ($diffAdd_S$) for *diffAdd* and Equations 5.1 – 5.3 show the the correctness results for the sub-modules in the combinational implementation.

Definition 5.1: *Specification for diffAdd*

$$diffAdd_S\ (i_1,\ i_2,\ b)\ o\ \hat{=}\ \ o = |i_1 - i_2| + b$$

$$
\begin{array}{lll}
sub_L\ (i_1,\ i_2)\ o & \Longrightarrow & o = i_1 - i_2 \qquad (5.1)\\
neg_L\ i\ o & \Longrightarrow & o = -i \qquad (5.2)\\
mux_L\ s\ (i_1,\ i_2)\ o & \Longrightarrow & o = \mathit{if}\ s\ \mathit{then}\ i_2\ \mathit{else}\ i_1 \qquad (5.3)\\
add_L\ (i_1,\ i_2)\ o & \Longrightarrow & o = i_1 + i_2 \qquad (5.4)
\end{array}
$$

The combinational logic implementation ($diffAdd_L$) is shown in Definition 5.2.

Definition 5.2: *Combinational logic implementation of diffAdd*

$$
\begin{array}{l}
diffAdd_L\ (i_1,\ i_2,\ b)\ o\ \hat{=}\\
\quad \exists s,\ d,\ n,\ x\ .\\
\qquad sub_L\ (i_1, i_2)\ (s,\ d)\ \&\\
\qquad neg_L\ d\ n\ \&\\
\qquad mux_L\ s\ (d,\ n)\ x\ \&\\
\qquad add_L\ (x,\ b)\ o
\end{array}
$$

The correctness goal for $diffAdd_L$ says that the implementation implies the specification (Equation 5.5).

$$\vdash \forall i_1,\ i_2,\ b,\ o.\ diffAdd_L\ (i_1,\ i_2,\ b)\ o \Longrightarrow diffAdd_S\ (i_1,\ i_2,\ b)\ o \qquad (5.5)$$

The first few steps of the proof of Equation 5.5 are shown in Figure 5.1. We begin by moving the implementation $diffAdd_L$ into the hypothesis list and then unfolding $diffAdd_L$ and the behavioral specification $diffAdd_S$. In the last step shown, we use Equations 5.1–5.4 to replace the implementation of each component of $diffAdd_L$ with its specification. At this point, standard mathematical reasoning can easily solve the goal.

5.2.2 Sequential Implementation with Register

Equations 5.6 – 5.10 show the correctness results for a simple register (delay element) and the sequential implementations of the components in *diffAdd*. We subscript the components by T (for "time") to signify that they are sequential models. The definitions are based on the combinational logic versions of the components.

$$
\begin{array}{lll}
reg_T\ B\ zB & \Longrightarrow & \forall t\ .\ (zB\ (t{+}1)) = (B\ t) \qquad (5.6)\\
sub_T\ (I_1,\ I_2)\ (S,\ D) & \Longrightarrow & \forall t\ .\ sub_L\ ((I_1\ t),\ (I_2\ t))\ ((S\ t),\ (D\ t)) \qquad (5.7)\\
neg_T\ D\ N & \Longrightarrow & \forall t\ .\ neg_L\ (D\ t)\ (N\ t) \qquad (5.8)\\
mux_T\ S\ (D,\ N)\ X & \Longrightarrow & \forall t\ .\ mux_L\ (S\ t)\ ((D\ t),\ (N\ t))\ (X\ t) \qquad (5.9)\\
add_T\ (I_1,\ I_2)\ (S,\ D) & \Longrightarrow & \forall t\ .\ add_L\ ((I_1\ t),\ (I_2\ t))\ ((S\ t),\ (D\ t)) \qquad (5.10)
\end{array}
$$

We combine the sequential components together to create a sequential implementation ($diffAdd_T$) in Definition 5.3. This implementation includes registers at the inputs to the

$\vdash\ \forall i_1,\ i_2,\ b,\ o.\ \mathit{diffAdd}_L\ (i_1,\ i_2,\ b)\ o \implies \mathit{diffAdd}_S\ (i_1,\ i_2,\ b)\ o$

BY `Begin proof`

- $\mathit{diffAdd}_L\ (i_1,\ i_2,\ b)\ o$

$\vdash\ \mathit{diffAdd}_S\ (i_1,\ i_2,\ b)\ o$

BY `Unfold` $\mathit{diffAdd}_L$`,` $\mathit{diffAdd}_S$

- $\mathit{sub}_L\ (i_1,\ i_2)\ (s,\ d)$
- $\mathit{neg}_L\ d\ n$
- $\mathit{mux}_L\ s\ (d,\ n)\ x$
- $\mathit{add}_L\ (x,\ b)\ o$

$\vdash\ o = |i_1 - i_2| + b$

BY `Equations 5.1 -- 5.4`
`(*` 'Implementations imply specifications' for sub_L, neg_L, mux_L, add_L `*)`

- $d = i_1 - i_2$
- $s = d < 0$
- $n = -d$
- $x = \mathit{if}\ s\ \mathit{then}\ n\ \mathit{else}\ d$
- $o = x + b$

$\vdash\ o = |i_1 - i_2| + b$

Figure 5.1: Beginning of proof of $\mathrm{diffAdd}_L$

circuit, and so in the goal (Equation 5.11) the output at time $(t{+}1)$ correspond to the inputs at time t.

Definition 5.3: *Sequential implementation of diffAdd*

$$
\begin{array}{l}
\mathit{diffAdd}_T\ (I_1,\ I_2,\ B)\ O \mathrel{\hat{=}} \\
\quad \exists zI_1,\ zI_2,\ zB,\ S,\ D,\ N,\ X : \mathit{int\ \ signal}\ . \\
\qquad \mathit{reg}_T\ I_1\ zI_1\ \ \&\ \ \mathit{reg}_T\ I_2\ zI_2\ \ \&\ \ \mathit{reg}_T\ B\ zB\ \ \& \\
\qquad \mathit{sub}_T\ (zI_1,\ zI_2)\ (S,\ D)\ \& \\
\qquad \mathit{neg}_T\ D\ N\ \& \\
\qquad \mathit{mux}_T\ S\ (D,\ N)\ X\ \& \\
\qquad \mathit{add}_T\ (X,\ zB)\ O
\end{array}
$$

$$
\begin{array}{l}
\vdash \forall I_1,\ I_2,\ B,\ O : \mathit{int\ \ signal}\ . \\
\quad \mathit{diffAdd}_T\ (I_1,\ I_2,\ B)\ O \implies \\
\qquad \forall t\ .\ \mathit{diffAdd}_S\ ((I_1\ t),\ (I_2\ t),\ (B\ t))\ (O\ (t{+}1))
\end{array}
\tag{5.11}
$$

5.3 Pipelined Implementation of DiffAdd

The pipeline (Figure 5.2) contains three stages: sub_P, neg_P, and add_P. The stages are built by adding registers and pipelining circuitry to sub_T, neg_T, and add_T. In the non-pipelined implementation ($\mathit{diffAdd}_T$) we used a multiplexer (mult_T) to select whether to use the output of sub_T or neg_T. In the pipeline we perform the analogous operation by either routing the transaction from sub_P directly to add_P or passing through neg_P and then on to add_P. This puts the functionality previously found in the multiplexer into the pipelining circuitry for sub_P and add_P. Another high level change is that we send the values of B in their transactions through sub_P and neg_P, while in the sequential model we sent B to only add_T.

5.3.1 Building Datapaths for Stages

We begin building the pipeline by adding support to route the value of B through the pipeline as shown in Definitions 5.4 – 5.6 and Figures 5.3 – 5.5. These circuits are the core datapaths for the stages. They are completely independent of how the four parameters to the framework will be instantiated.

$$
\begin{array}{llcll}
\mathit{sub}_D\ (I_1,\ I_2,\ B) & (S,\ D,\ B_s) & \hat{=} & \mathit{sub}_T\ (I_1,\ I_2)\ (D,\ S) & \&\ \ \mathit{wire}_T\ B\ B_s \\
\mathit{neg}_D\ (D,\ B_s) & (N,\ B_n) & \hat{=} & \mathit{neg}_T\ D\ N & \&\ \ \mathit{wire}_T\ B_s\ B_n \\
\mathit{add}_D\ (D,\ B_x) & O & \hat{=} & \mathit{add}_T\ (D,\ B_x)\ O &
\end{array}
$$

(5.12)
(5.13)
(5.14)

We are implementing this pipeline in bottom-up fashion; we begin with the datapaths of the stages. We create a combinational representation of the datapath for each pipeline stage by replacing each sequential component in the datapath with its corresponding combinational

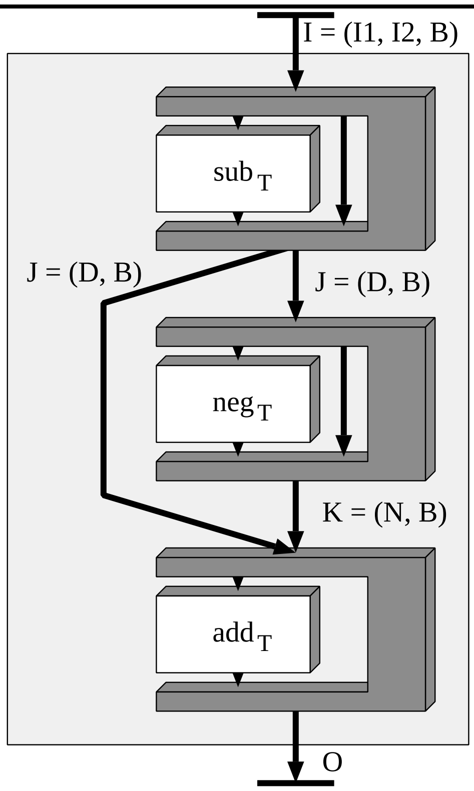


Figure 5.2: Diagram of pipeline for *diffAdd*

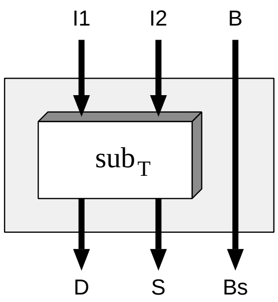


Figure 5.3: sub_D

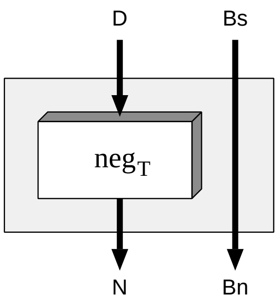


Figure 5.4: neg_D

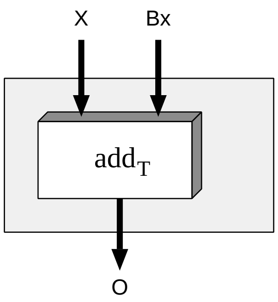


Figure 5.5: add_D

Time-domain datapaths

logic implementation. For example, in Equation 5.15 we replace sub_T in Equation 5.12 with sub_L and replace the wire connecting B and B_s with an equality predicate. We use lowercase variables in Equations 5.15 – 5.17, because the circuits are relations over values and not signals.

$$sub_M\ (i_1,\ i_2,\ b)\ (s,\ d,\ b_s) \ \hat{=}\ sub_L\ (i_1,\ i_2)\ (s,\ d)\ \ \&\ \ b_s = b \tag{5.15}$$
$$neg_M\ (d,\ b_s)\ \ (n,\ b_n) \ \hat{=}\ neg_L\ d\ n\ \ \&\ \ b_n = b_s \tag{5.16}$$
$$add_M\ (d,\ b)\ \ o \ \hat{=}\ add_L\ (d,\ b)\ o \tag{5.17}$$

5.3.2 Instantiating Framework Parameters

Having designed the datapaths for the stages, we now build the pipelining circuitry for the stages. This corresponds to the dark gray boxes surrounding each datapath in Figure 5.2. We begin by choosing instantiations for the four parameters to the framework (protocol, arbitration, control, and ordering).

To guide the process of choosing instantiations and verifying their correctness, we use the decision trees for the parameters (Figures 4.1, 4.2, 4.3, and 4.4). The following paragraphs describe the paths we take through the decision trees; the resulting instantiations are summarized in Table 5.1.

Protocol We constrain the bottom environment to immediately accept requests, but we do not constrain the top environment to issue a new transaction every cycle. So we incur the obligation that stages must accept requests if they are not busy. The topology of the pipeline is not a tree, because the paths from *sub* and *neg* merge at *add*. We decide that neither the environment nor the entry stage (*sub*) is responsible for preventing collisions inside the pipeline. There is only one entry stage, but the latency to the join at *add* is different for the two paths, so we require a general protocol scheme. For a general protocol scheme we must define request and accept virtual signals between any two stages that can transfer transactions, between the top environment and the entry stage, and between the exit stage and the bottom environment.

Arbitration The pipeline does not have a tree topology, and we were unable to use a transit protocol scheme, but the topology is acyclic. We now check if the topology satisfies the rules for an static prioritization scheme. There are no overlapping skips. The only stage that can face contention is *add*, which can receive requests from both *sub* and *neg*. The path from *sub* to *add* skips over *neg*. The static arbitration scheme requires that non-skipping transactions have priority. If we give priority to *neg* over *sub* when contending for *add*, then we can use a static arbitration scheme.

Control The topology of the pipeline is acyclic, so the pipeline uses a no-loops control scheme.

Ordering There is only one entry stage in the pipeline. There is at most one transaction in each of the three stages, so clearly transactions exit each stage in the same order that they enter it. Not all transactions take the same path through the pipeline, but there is only one exit stage. The two different paths to *add* have different latencies. Now, we must decide if transactions enter stages in the same order that they enter the pipeline. Clearly this is true for *sub* and *neg*. To check this condition for *add* we need to determine if a transaction (T_s) that skips *neg* can pass a transaction (T) that did not skip *neg*. In order for the in-order ordering scheme to be violated, T must enter the pipeline one or more cycles before T_s, but enter *add* after T_s. If T enters exactly one cycle before T_s, then T will be in *neg* at the same time that T_s is in *sub*. If we give *neg* a higher priority than *sub*, then T will have priority over T_s and will enter *add* before T_s. If T enters more than one cycle before T_s, then T will enter *add* before T_s requests *add*. Thus transactions enter stages in the same order that they enter the pipeline and we can use an in-order ordering scheme.

Table 5.1: Framework instantiations for pipeline

Protocol	general
Arbitration	static priorities
Control	topological
Ordering	in-order

5.3.3 Building the Core for Each Stage: Generating Request and Accept Signals

The core of a stage is its datapath and whatever circuitry is required to generate the request and accept virtual signals. The implementations of this circuitry is determined by the class of the stage. We need to determine the class of each of the three stages. We cannot guarantee that any of the stages receive a request every cycle, so none of the stages are degenerate. We have constrained the bottom environment to accept requests immediately. This guarantees that *add* cannot stall, so it is a “unit-delay” stage. To check if *neg* can stall, we need to know if it can ever have a request denied. The request from *neg* is the highest priority request to a stage that cannot stall. This is sufficient to guarantee that *neg* cannot stall, and so it is a “unit-delay” stage. It is easy to confirm that *sub* can stall: if a transaction that skips *neg* follows a transaction that uses *neg*, then the second transaction will stall in *sub*. Thus, *sub* is a “general” stage.

The core for each stage with request and accept generation circuitry is shown in Definitions 5.4 – 5.6. These cores contain the datapaths from Definitions 5.4 – 5.6 and the circuitry required to generate request and accept signals. The datapaths for the two stages that cannot stall (*add* and *neg*) do not need accept signals, use a register to latch in the current value of their input request signal and generate their output request, and use the

Table 5.2: Instantiations for datapaths for stages in diffAdd

sub	General
neg	Unit-delay
add	Unit-delay

matchDelay matching relation with a delay of one. For *sub*, we use signals for the both the request and the accept virtual signals and use *mkReqAcc*, the standard request/accept state machine (Equation 4.5). The matching relation for *add* is *matchUnique*.

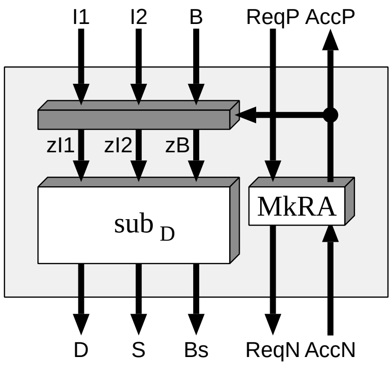


Figure 5.6: sub_C

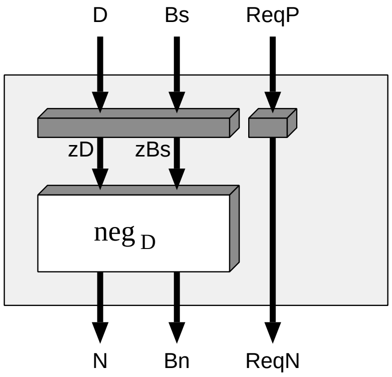


Figure 5.7: neg_C

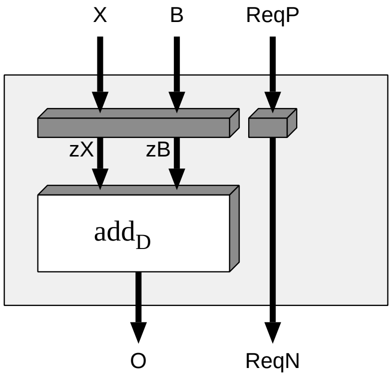


Figure 5.8: add_C

Cores of stages

Definition 5.4: *Core of subtract stage*

$$
\begin{array}{l}
sub_C \; ((I_1,\ I_2,\ B),\ (ReqP,\ AccP)) \;\; ((S,\ D,\ B_s),\ (ReqN,\ AccN)) \;\; \hat{=} \\
\quad \exists zI_1,\ zI_2,\ zB\ . \\
\qquad regLd_T\ AccP\ (I_1,\ I_2,\ B)\ (zI_1,\ zI_2,\ zB)\ \& \\
\qquad sub_D\ (zI_1,\ zI_2,\ zB)\ (D,\ S,\ B_s) \qquad\quad \& \\
\qquad mkReqAcc\ (ReqP,\ AccP)\ (ReqN,\ AccN)
\end{array}
$$

Definition 5.5: *Core of negate stage*

$$
\begin{array}{l}
neg_C\ ((D,\ B_s),\ ReqP)\ \ ((N,\ B_n),\ ReqN)\ \ \hat{=} \\
\quad \exists zD,\ zB_s\ . \\
\qquad (reg_T\ D\ zD)\ \ \&\ \ (reg_T\ B_s\ zB_s)\ \ \& \\
\qquad neg_D\ (zD,\ zB_s)\ (N,\ B_n)\qquad\qquad \& \\
\qquad reg_T\ ReqP\ ReqN
\end{array}
$$

Definition 5.6: *Core of addition stage*

$$
\begin{array}{l}
add_C\ ((X,\ B),\ ReqP)\ \ (O,\ ReqN)\ \ \hat{=} \\
\quad \exists zX,\ zB\ . \\
\qquad (reg_T\ X\ zX)\ \ \&\ \ (reg_T\ B\ zB)\ \& \\
\qquad add_D\ (zX,\ zB)\ O\qquad\qquad \& \\
\qquad reg_T\ ReqP\ ReqN
\end{array}
$$

5.3.4 Building Stages: Interface, Arbitration and Control Circuitry

Having chosen instantiations for the four parameters and built the cores with request and accept generation circuitry, we proceed with the interface, arbitration, and control circuitry for the stages. We begin by defining the request and accept virtual signals between each pair of connected stages. Then we choose how to provide the required functionality for the pipelining circuitry for the stages. In the discussion that follows *TOP* and *Bot* refer to the ports between the pipeline and a top and a bottom environment. The pipeline is independent of the environment; we use *TOP* and *Bot* only as identifiers to reference these ports.

5.3.4.1 Virtual Signals Between Stages

The framework defines four pieces of information that can be transferred between stages: data-stationary control, data, hierarchical information (used primarily to pass data-stationary control information through hierarchical pipelines), requests, and accepts. There is no data stationary control used in *diffAdd*. The datapaths for the stages define the data that is transferred between stages. There is no hierarchical information transferred between stages. Table 5.3 shows the implementations for the request and accept virtual signals for the five inter-stage connections in *diffAdd*.

In accordance with the classes of the three stages, the request virtual signals for all five connections between stages are implemented in hardware as the signals *reqTop*, *reqSubAdd*, *reqSubNeg*, *reqNegAdd*, and *reqBot*. The requests from *TOP* to *sub* and *sub* to *add* may not be immediately accepted, so these accept virtual signals will be implemented in hardware. The other three requests will always be immediately accepted, so their accept virtual signals

Table 5.3: Instantiations for request and accept signals between stages for *diffAdd*

	TOP→*sub*	*sub*→*add*	*sub*→*neg*	*neg*→*add*	*add* →*Bot*
Req	*reqTop*	*reqSubAdd*	*reqSubNeg*	*reqNegAdd*	*reqBot*
Acc	*accTop*	*accSubAdd*	*true*	*true*	*true*

do not require any hardware and are implemented using the constant *true*.
The stages *add* and *neg* can not stall, so their accept signals are always true. The stage *sub* can stall, but only when requesting *add*. Requests from *sub* to *neg* will be accepted immediately. This is because the only source stage for *neg* is *sub*, and *neg* can contain at most one transaction. This means that we can implement the accept signal from *sub* to *neg* using the constant *true*. This is an example of how the framework specifies required *functionality* but does not restrict the hardware that is used to achieve the functionality.
The implementations of the functional units for arbitration, interface, and control are shown in Table 5.4.

Table 5.4: Instantiations of segment components for stages

	Arb	Prev	InterfaceP	MkOp	MkNext	Next	InterfaceN
sub	—	—	—	—	*Next*=*S*	0=*add* 1=*neg*	2-1 mux
neg	—	—	—	—	—	—	—
add	*Prev*=*reqNeg*	0=*sub* 1=*neg*	2-1 mux	—	—	—	—

If a stage can receive inputs from more than one source, then we need to arbitrate between requests, assign values of the *Prev* signal to the set of previous stage and implement interface circuitry. The only stage in *diffAdd* that has multiple potential sources is *add*, which can receive inputs from *sub* or *neg*.
In accordance with the static prioritization scheme, *add* gives higher priority to requests from *neg* than from *sub*. When we have exactly two previous stages, we can use slightly more efficient hardware if we assign *Prev*=1 for the higher priority stage and *Prev*=0 for the lower priority stage. So we assign *Prev*=1 for *neg* and use the standard implementation of 2-1 multiplexers and a 1-2 demultiplexer for InterfaceP. The other two stages do not require any interface circuitry to previous stages.
Each of the datapaths in *diffAdd* is capable of performing a single operation, so no hardware is needed for the MkOp functional unit, which chooses the operation a datapath is to perform.

The MkNext circuit determines the next stage to go to and generates any data-stationary control for the transaction. This pipeline does not have any data-stationary control. The stage sub_P uses the value of S to determine whether to send its transaction to *neg* or to *add*. With only two possible next stages, we can encode the next stage directly using S, and so do not need any additional hardware. The stage sub_P uses the standard InterfaceN circuitry of a 2-1 multiplexer and 1-2 demultiplexer to select which accept virtual signal (*accSubAdd* or *accSubNeg*, which simplified to *true*) to examine.

5.3.4.2 Optimizations of Hardware

We now step through the process of building the pipeline stage sub_P. This example illustrates how we refine an implementation to achieve an optimized implementation without sacrificing confidence in correctness or redoing any verification. We begin with the datapath sub_C and the standard interface circuitry: a 1-2 demultiplexer and a 2-1 multiplexer. The 2-1 multiplexer is used to select which accept signal to route to *accN*. If *accN* is true, then the stage can transfer the transaction to the desired next stage (*add* or *neg*). If S is true, then the transaction wishes to transfer to *neg* and so we connect the accept signal from *neg* to *accN*. Otherwise, we connect the accept signal from *add*.

We make use of knowledge about the request and accept virtual signals to replace the 2-1 multiplexer with a simple two gate circuit. The standard multiplexer based implementation is shown in Equation 5.18. The signals *xAccN*[0] and *xAccN*[1] are used to connect the stage to the other stages in the pipeline. The equivalent behavioral description of this circuit is shown in Equation 5.19. From Table 5.4 we know that the indices for the next stages are *"add"*=0 and *"neg"*=1. So, we substitute the signals *accAdd* for *xAccN*[0] and *accNeg* for *xAccN*[1]. The accept signal from *neg* (*accNeg*) is always true, so we substitute *true* for *accNeg* in Equation 5.21. We implement the *if-then-else* in Equation 5.21 with just an OR gate, as shown in Equation 5.22.

$$mux_T\ S\ (xAccN[0],\ xAccN[1])\ accN \tag{5.18}$$
$$accN = \mathit{if}\ S\ \mathit{then}\ xAccN[1]\ \mathit{else}\ xAccN[0] \tag{5.19}$$
$$accN = \mathit{if}\ S\ \mathit{then}\ accNeg\ \mathit{else}\ accAdd \tag{5.20}$$
$$accN = \mathit{if}\ S\ \mathit{then}\ true\ \mathit{else}\ accAdd \tag{5.21}$$
$$accN = S\ \vee\ accAdd \tag{5.22}$$

In this section, we demonstrated how we used knowledge about the request and accept signals between stages to optimize the interface and arbitration hardware for sub_P. Using techniques similar to those just shown, we simplify the InterfaceP and Arbitration circuitry for add_P to an OR gate, an inverter, and two multiplexers.

5.3.4.3 Final Implementations of Stages

The final implementations of the stages are shown in Figures 5.9 – 5.11. The implementations of sub_P and add_P use the optimized interface hardware and from Table 5.4, we know that neg_P does not require any additional interface, arbitration, or control hardware.

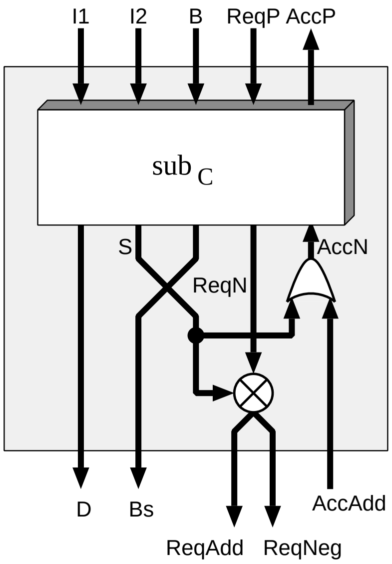


Figure 5.9: sub_P

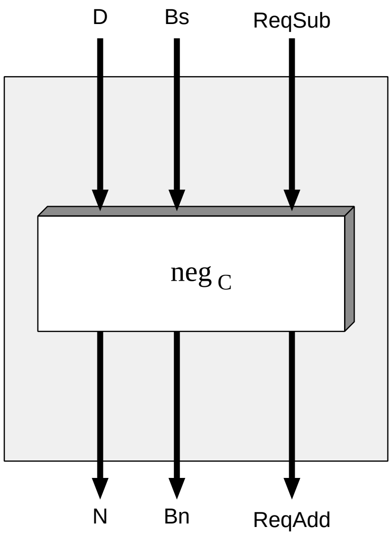


Figure 5.10: neg_P

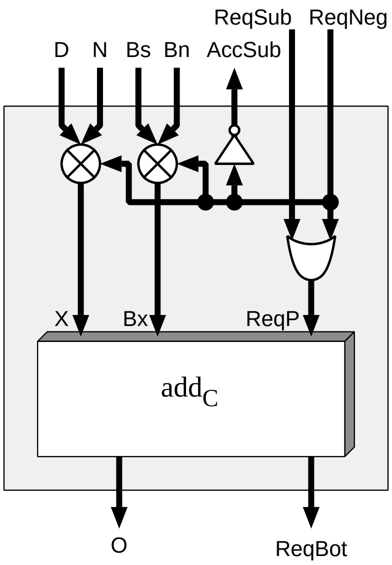


Figure 5.11: add_P

Complete stages

The complete pipeline is shown in Definition 5.7 and Figure 5.12. The pipeline ($diffAdd_P$) has inputs for data (I), request ($reqTop$), and accept ($accTop$) signals and outputs for resulting data (O) and request ($reqBot$) signals. Internally the stages are connected with two data signals (J and K), three request signals, and one accept signal. The body of the pipeline contains the three stages (sub_P, neg_P, and add_P).

Definition 5.7: *Pipelined implementation of diffAdd*

$$
\begin{array}{lll}
\multicolumn{3}{l}{diffAdd_P\ (I,\ (reqTop,\ accTop))\ (O,\ reqBot) \hat{=}} \\
\multicolumn{3}{l}{\quad \exists J,\ K,\ reqSubNeg,\ reqSubAdd,\ reqNegAdd,\ accSubAdd\ .} \\
\quad\quad sub_P & (I,\ (reqTop,\ accTop)) & \\
 & (J,\ (reqSubAdd,\ reqSubNeg,\ accSubAdd)) & \& \\
\quad\quad neg_P & (J,\ reqSubNeg) & \\
 & (K,\ reqNegAdd) & \& \\
\quad\quad add_P & ((J,\ K),\ (reqSubAdd,\ reqNegAdd,\ accSubAdd)) & \\
 & (O,\ reqBot) &
\end{array}
$$

Figure 5.12 shows the pipeline with the internal components of the stages exposed. The pipelining circuitry for *sub* consists of a register, the simple state machine to generate requests and accepts, a two input OR gate and a 1-2 multiplexer. The only pipelining circuitry for *neg* are the registers for the data and request signal. For *add*, we added a register, two multiplexers, an inverter, and a two-input OR gate. The total amount of circuitry required to turn the non-pipelined implementation of *diffAdd* into a pipeline is quite small. Furthermore, the framework did not require that any additional hardware be implemented beyond that which an ad-hoc design process would have led to.

5.3.5 Combinational Representation and Matching Relation for Pipeline

To create the combinational representation of the pipeline, we compose the combinational representations of the stages according to the paths that transactions can take through the pipeline. In the pipeline, the control circuitry in *sub* chooses to route the transaction to either *add* or *neg* based on the value of S. This is done using the MkNext circuitry for the stage (Table 5.4). The equivalent behavior of this circuit is: *if S then Next= "neg" else Next= "add"*. Translated into a path in the combinational representation, this results in the the *if-then-else* after sub_M (Definition 5.8).
From the decision tree for ordering schemes, we know that we can use the in-order matching relation (Definition 4.4) for the pipeline.

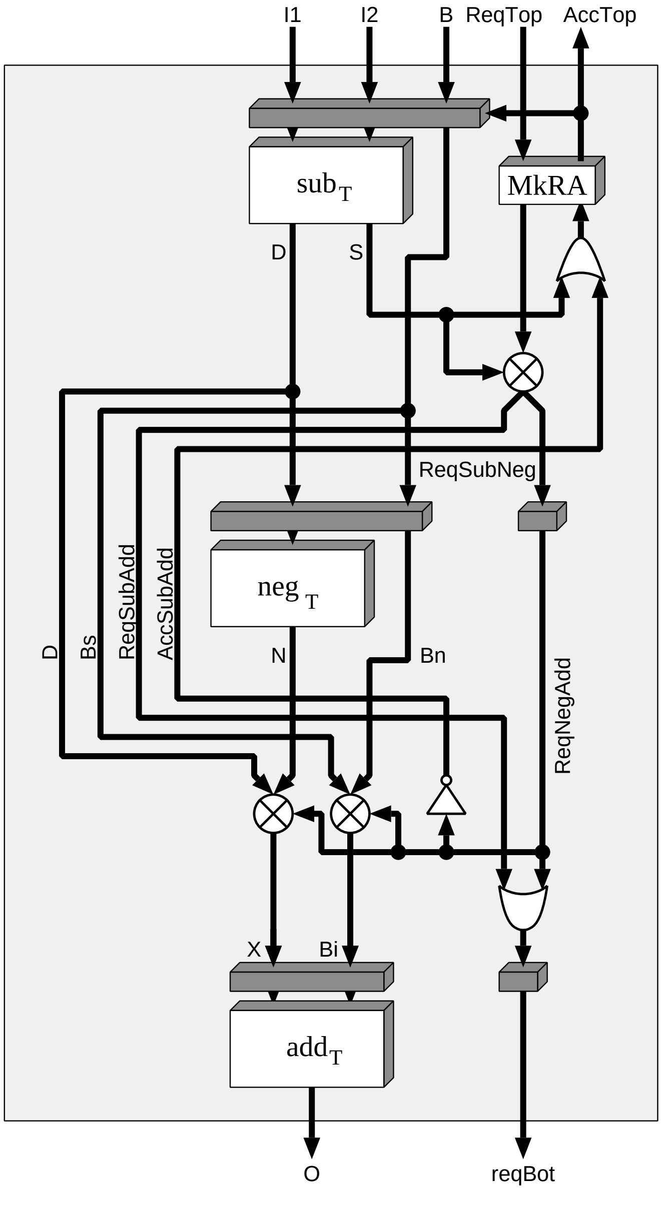


Figure 5.12: Detailed diagram of pipeline for *diffAdd*

Definition 5.8: *Combinational representation of diffAdd*

$$
\begin{array}{l}
\mathit{diffAdd}_M\ (i_1,\ i_2,\ b)\ o\ \hat{=} \\
\quad \exists s,\ d,\ b_s,\ n,\ b_n\ . \\
\quad\quad \mathit{sub}_M\ (i_1,\ i_2,\ b)\ (s,\ d,\ b_s)\ \& \\
\quad\quad \mathit{if}\ s \\
\quad\quad \mathit{then}\ \ \mathit{neg}_M\ (d,\ b_s)\ (n,\ b_n)\ \& \\
\quad\quad \ \ \mathit{add}_M\ (n,\ b_n)\ o \\
\quad\quad \mathit{else}\ \ \mathit{add}_M\ (d,\ b_s)\ o
\end{array}
$$

5.4 Verification of Pipelined Implementation of DiffAdd

As was briefly described in Section 2.4, we begin the proof of correctness for $\mathit{diffAdd}_P$ (Figure 5.13) by separating the functionality from the pipelining concerns using Theorem 2.1.

$$\vdash\ (\mathit{diffAdd}_P,\ \mathit{matchInOrder}) \sqsupset \mathit{diffAdd}_S$$

BY `Theorem 2.1 using` $\mathit{diffAdd}_M$
(∗ Transitivity of *satisfies* ∗)

$$
\begin{array}{l}
\vdash\ \forall\ I_1,\ I_2,\ B,\ O\ . \\
\quad \mathit{diffAdd}_M\ (I_1,\ I_2,\ B)\ O \implies \mathit{diffAdd}_S\ (I_1,\ I_2,\ B)\ O \\
\vdash\ (\mathit{diffAdd}_P,\ \mathit{matchInOrder}) \sqsupset \mathit{diffAdd}_M
\end{array}
$$

Figure 5.13: First proof step in verification of $\mathit{diffAdd}_P$

In the second step of the proof (Figure 5.14 we apply the main theorem for the framework (Theorem 2.2), which decomposes the proof obligation that the pipeline *satisfies* the combinational representation into four simpler goals.

The first subgoal in Figure 5.14 is to show that each stage *satisfies* its combinational representation, the second goal requires proving that the constraints of the stages are met, the third subgoal is for the framework parameters, and the last subgoal is to show that the combinational representation implies the specification.

Verifying that a stage *satisfies* its combinational representation is much simpler than proving the same for a pipeline, because a stage can contain at most one transaction. We constructed the core for each stage (sub_C, neg_C, and add_C) from the same sequential implementation (sub_T, neg_T, and add_T) that the combinational representations (sub_M, neg_M, and add_M) are based on. This means that the the I/O functionality of each combina-

$\vdash$ ($diffAdd_P$, $matchInOrder$, $nullConstraints$) $\sqsupset$ $diffAdd_M$

BY `Theorem 2.2`

$\vdash$ $CoresOk\ diffAdd_P$
$\vdash$ $ConstraintsOk\ diffAdd_P\ nullConstraints$
$\vdash$ $CombOk\ diffAdd_P\ diffAdd_M$
$\vdash$ $FworkParamsOk$ ($diffAdd_P$, $matchInOrder$, $nullConstraints$)

Figure 5.14: Pipelining proof step in verification of $diffAdd_P$

tional representation is equivalent to that of the datapath. We now have to show that the transactions terminate from the stages according to the matching relations. We built the request/accept generation and data-storage circuitry and chose the matching relations using the standard implementations shown in Table 5.2. The correctness of these implementations for *diffAdd* only relies upon our proofs that add_P and neg_P can not stall, which was given in Section 5.3.3. Thus, we know that the stages are modeled correctly.

For the second subgoal from Figure 5.13 we show that the pipeline conforms to the specifications for the parameters. Our choices for the instantiations of the parameters were guided by the decision trees (Section 5.3.2) and we incurred verification obligations while traversing the trees. To verify that the protocol specification is satisfied, we need to show three things. First we must show that the data signals and request/accept signals are connected correctly between stages. These rely on checking the interface circuitry for the stages and the indexing for previous and next stages (Table 5.4). Second, when two stages exchange a request and accept, they must transfer a transaction. Third, if a request is not accepted, then it must be repeated in the next clock cycle.

We solve the subgoals for the protocol scheme using the definitions in table Table 5.4. From *CoresOk* we know that the stages generate request and accept signals and store transactions correctly. In Table 5.4 we used standard implementations for the interface functional units, so we know that the signals get routed between the datapaths of the stages correctly. We optimized the interface between *add* and *neg* using the fact that the request from *add* to *neg* is always accepted immediately.

There are three parts to the topological-rules for an acyclic static-priority arbitration scheme. First, the topology of the pipeline must be acyclic, which it obviously is. Second, there cannot be any overlapping skips. This is easily satisfied, because there is only one skip in the pipeline: transactions going from *sub* to *add* skip over *neg*. Third, transactions that do not skip have higher priority than skipping transactions when contending for a stage. Transactions from *neg* have higher priority than those from *add*.

There are two requirements for a control scheme that is checked using topological rules.

First, the topology must be acyclic. This is known from the arbitration scheme verification, which required that the pipeline be acyclic. Second, the implementation of the control logic must follow that in the combinational representation. The only control decision in the pipeline is whether to send a transaction to neg_P or add_P. We check that this condition is satisfied using the implementation of *MkNext* and the *next* and *Prev* signals for add_P shown in Table 5.4.

We used an in-order ordering scheme. Based upon the path we took through the decision tree, we need to show that transactions enter stages in the order that they enter the pipeline and that they exit each stage in the order that the enter it. In Section 5.3.2 we argued that transactions enter the stages in the same order that they enter the pipeline.

By showing that the pipeline meets the specifications for the framework parameters, we complete the proofs of the subgoals shown in Figure 5.14. This completes the proof of the second subgoal in Figure 5.13, which leaves us with showing that the combinational representation ($diffAdd_M$) implies the specification (Figure 5.15). After unfolding the definitions and replacing the combinational representations with their specifications, we are left with a sequent which is almost identical to the final sequent shown for the combinational logic implementation of *diffAdd* (Figure 5.15). The only difference between the two sequents is the *if-then-else*. The subsequent reasoning for the two sequents would only require doing a case split on whether i_1 is greater than i_2 and some standard arithmetical reasoning.

5.5 Conclusion

In this chapter we stepped through the design and verification of a simple pipeline that contained a structural hazard. We began with a purely combinational implementation and showed how the framework is used to guide the process of creating a verified and efficient pipelined implementation. We began by using the decision trees to find instantiations for the four framework parameters and then determined the class of each stage. This allowed us to use standard implementations for the request/accept generation circuitry and the interface, arbitration and control circuitry. Because we had used the framework and standard pipelining circuitry we were able to reduce the verification of the pipeline to a proof about a non-pipelined description that was very similar to the combinational implementation shown at the beginning of the chapter.

$\vdash\ \forall i,\ o\ .\ \mathit{diffAdd}_M\ i\ o \implies \mathit{diffAdd}_S\ i\ o$

BY `Unfold` $\mathit{diffAdd}_M$ `,` $\mathit{diffAdd}_S$

- $\mathit{sub}_M\ (i_1,\ i_2,\ b)\ (s,\ d,\ b_s)$
- $\mathit{if}\ s$
  $\mathit{then}\ \ \mathit{neg}_M\ (d,\ b_s)\ (n,\ b_n)$ &
  $\qquad \mathit{add}_M\ (n,\ b_n)\ o$
  $\mathit{else}\ \ \mathit{add}_M\ (d,\ b_s)\ o$

$\vdash\ o = |\ i_1 - i_2\ | + b$

BY `Equations 5.15 -- 5.17 (*` *Unfold definitions of* sub_M, neg_M, add_M `*)`

- $d = i_1 - i_2$
- $s = (d < 0)$
- $b_s = b$
- $\mathit{if}\ s$
  $\mathit{then}\ n = -d$ & $b_n = b_s$ &
  $\qquad o = n + b_n$
  $\mathit{else}\ \ o = d + b_s$

$\vdash\ o = |\ i_1 - i_2\ | + b$

BY `Substitute using` $b_s = b$ `and` $b_n = b_s$

- $d = i_1 - i_2$
- $s = (d < 0)$
- $\mathit{if}\ s\ \mathit{then}\ n = -d$ &
  $\qquad o = n + b$
  $\mathit{else}\ \ o = d + b$

$\vdash\ o = |\ i_1 - i_2\ | + b$

Figure 5.15: Proof steps for combinational representation

Chapter 6

Adirondack Microprocessor: Overview of a Large Example

6.1 Introduction

This chapter gives an overview of the Adirondack microprocessor, a fictitious RISC super-scalar super-pipelined microprocessor, and how the framework is used in its design and verification. The Adirondack (ADK) is based on recent microprocessors from Digital Equipment Corporation, primarily the DEC Alpha 21064 (the first generation Alpha) [McL93] and the Vax 8600 Model 200 (Also known as the NVAX) [BBC+89]. Our purpose is to show how we can use the framework to characterize and verify complex pipelines. Because of the size of this example and our emphasis on the pipelining aspects of the design, we do not describe the design or verification of the datapath.

We begin with an overview of the ADK (Section 6.2) We cover the instruction pipeline by first describing a single-issue instruction pipeline (Section 6.3). We then show the dual-issue instruction pipeline used in the ADK (Section 6.4) and demonstrate how the framework can be used to reduce the verification of the datapath of this super-scalar pipeline to that of the single-issue instruction pipeline. Next we describe the floating-point pipeline (Section 6.5). We conclude by describing how the individual ADK pipelines can be combined together to create the complete microprocessor (Section 6.6).

6.2 Microprocessor Overview

Figure 6.1 shows the block diagram of the Adirondack microprocessor. There are five main pipelines. The dual-issue instruction pipeline at the top fetches, decodes, and issues instructions. Below the instruction pipeline are four execution pipelines: the integer, load/store, branch, and floating-point pipelines. The integer execution pipeline performs integer arithmetic and logical operations. The load/store pipeline acts as the interface between the integer and floating-point register files and memory. The branch unit decides whether the branch was predicted correctly in the instruction pipeline. The floating-point

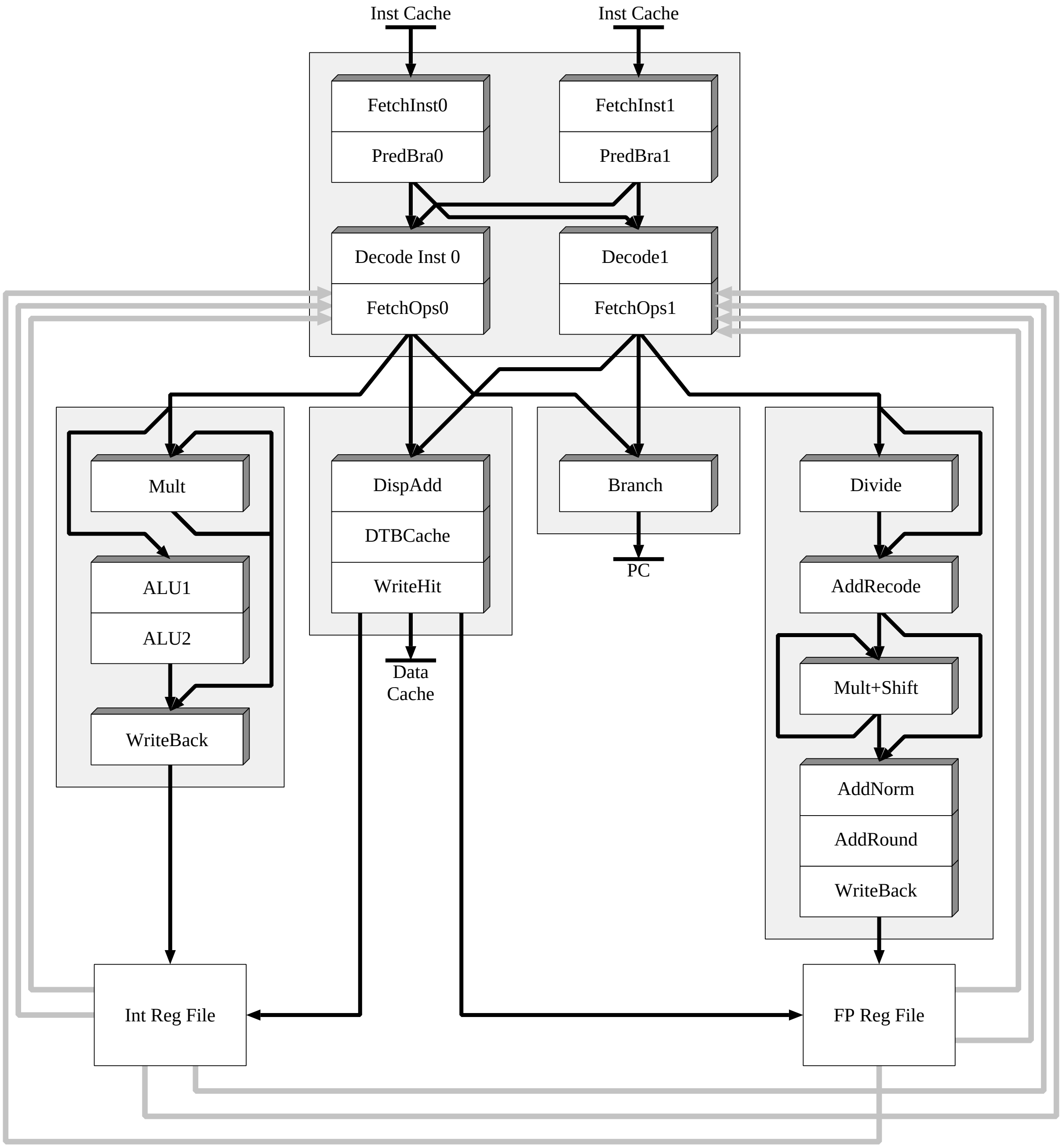


Figure 6.1: Block diagram of Adirondack microprocessor

pipeline performs floating-point arithmetic operations. The integer and floating-point register files are shown below the execution pipelines. The black lines indicate the paths taken by transactions. The gray lines indicate the flow of data operands. The overall structure, integer pipeline, load/store pipeline, and instruction pipeline shown in this figure are based on the DEC Alpha 21064. The floating-point pipeline is based on the VAX 8600 model 200.
The framework describes how to reason about and handle structural hazards. A microprocessor also has to handle data and control hazards. Preliminary work on adding data and control hazards to the framework appears quite promising: we believe that they can be added onto the existing framework without affecting the current modeling of structural hazards. The ideas are briefly outlined in Chapter 9.

6.3 Single-Issue Instruction Pipeline

The instruction pipeline shown in Figure 6.2 is representative of a very simple pipeline. It is half of the standard dual-issue instruction pipeline for the ADK, described in Section 6.4. The four stages are: Fetch Instruction (FetchInst), Predict Branch (PredBra), Decode Instruction (Decode), and Fetch Operands (FetchOps). Each stage is capable of performing only one operation and sends its output transactions to only one stage. We further simplify this pipeline by constraining the environment to immediately accepts requests and to schedule transactions in the input stream so that there are no data or control hazards between transactions. The MIPS R3000 [Kan88] uses these same two constraints.

6.3.1 Instantiations for Framework Parameters

We use the decision trees (Figures 4.1, 4.2, 4.3, and 4.4) to choose instantiations for the four framework parameters. Table 6.1 shows the instantiations for the framework parameters.

Protocol The environment always accepts requests immediately, but does not issue new transactions every cycle, so we incur the obligation that stages must accept a request if they are not busy. The pipeline has a tree topology and all stages have a latency of one, so we end up with a transit protocol scheme.

Arbitration The pipeline has a tree topology, so we have a degenerate arbitration scheme.

Control The topology is a directed acyclic graph, so we meet the specification for the topological rules.

Ordering There is only one entry stage and transactions exit each stage in the order they enter the stage. All transactions take the same path through the pipeline, all stages have a latency of one cycle, and the bottom environment accepts requests immediately. We have met the requirements for a uniform-latency ordering scheme with a latency of four.

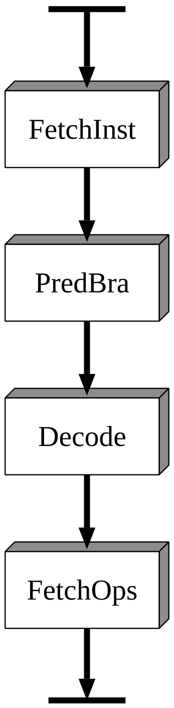


Figure 6.2: Block diagram of instruction pipeline

Table 6.1: Instantiations for framework parameters for single-issue instruction pipeline

Protocol	Degenerate
Arbitration	Degenerate
Control	Topological rules
Ordering	Uniform latency

6.3.2 Building the Pipeline

6.3.2.1 Generating Requests and Accepts for Stages

We begin the process of building the pipelining circuitry for the stages by adding hardware to generate request and accept virtual signals to the datapaths. We do this by finding the class of each stage, (degenerate, unit-delay, or general) as described in Section 4.3.4. We are using a transit protocol scheme, so we know that the stages are all unit-delay stages (Table 6.2). The core of a unit-delay stage uses a register that latches in new data on each cycle and a register to store the request signal for stage. Unit-delay stages use the *matchDelay* matching relation with a delay of one cycle. In accordance with the transit protocol scheme we use request signals between each pair of stages and the accept virtual signals are always true.

Table 6.2: Instantiations for stage classes in single-issue instruction pipeline

FetchInst	Unit-delay
PredBra	Unit-delay
Decode	Unit-delay
FetchOps	Unit-delay
Decode	Unit-delay

6.3.2.2 Building Interface, Arbitration and Control Hardware

There is no control information used in the pipeline and each stage is connected to only one source and one destination stage, so there is no need for interface, arbitration or control circuitry

6.3.2.3 Combinational Representation and Matching Relation

The combinational representation of the pipeline ($InstSingle_M$) sends transactions through each of the stages (Definition 6.1). Because we are using a uniform-latency ordering scheme with a latency of four, we use the *matchDelay* matching relation with a delay of four (Definition 6.2).

6.3.3 Verification

The are two purposes for including this example. The first is to demonstrate how the framework can be used with a very simple pipeline. The second is related to Section 6.4, where we describe the dual-issue instruction pipeline for the ADK. The dual-issue pipeline is essentially two copies of the pipeline from this section executing in parallel. Using the

Definition 6.1: *Combinational representation – single-issue instruction pipeline*

$$
\begin{array}{l}
\mathit{InstSingle}_M \;\; \mathit{TransTop} \;\; \mathit{TransBot} \;\; \hat{=} \\
\quad \exists\, \mathit{TransFi},\ \mathit{TransPb},\ \mathit{TransDc}\ . \\
\qquad \begin{array}{llll}
\mathit{FetchInst} & \mathit{TransTop} & \mathit{TransFi} & \& \\
\mathit{PredBranch} & \mathit{TransFi} & \mathit{TransPb} & \& \\
\mathit{Decode} & \mathit{TransPb} & \mathit{TransDc} & \& \\
\mathit{FetchOps} & \mathit{TransDc} & \mathit{TransBot} &
\end{array}
\end{array}
$$

Definition 6.2: *Matching relation – single-issue instruction pipeline*

$$\mathit{matchInstSingle} \;\; \hat{=} \;\; \mathit{matchDelay}\ 4$$

framework we can reduce a proof about the functionality of the dual-issue pipeline to that of this single-issue pipeline.

To aid in this comparison, we define *InstDataSpec* (Definition 6.3) as the specification for the input/output relationship between corresponding transactions.

Definition 6.3: *Specification for Data in Instruction Pipeline*

$$
\begin{array}{l}
\mathit{InstDataSpec}\ T_i\ T_o \ \hat{=} \\
\quad \mathit{case}\ ((\mathit{class} \circ \mathit{opcode})\ T_i) \\
\qquad \begin{array}{llll}
\mathit{of}\ \mathit{IntBr} & \Rightarrow & \mathit{InstIntBrSpec}\ T_i\ T_o & (*\text{integer branch}*) \\
|\ \ \mathit{IntOp} & \Rightarrow & \mathit{InstIntOpSpec}\ T_i\ T_o & (*\text{integer operation}*) \\
|\ \ \mathit{IntLd} & \Rightarrow & \mathit{InstIntLdSpec}\ T_i\ T_o & (*\text{integer load}*) \\
|\ \ \mathit{IntSt} & \Rightarrow & \mathit{InstIntStSpec}\ T_i\ T_o & (*\text{integer store}*) \\
|\ \ \mathit{FpBr} & \Rightarrow & \mathit{InstFpBrSpec}\ T_i\ T_o & (*\text{floating}-\text{point branch}*) \\
|\ \ \mathit{FpOp} & \Rightarrow & \mathit{InstFpOpSpec}\ T_i\ T_o & (*\text{floating}-\text{point operation}*) \\
|\ \ \mathit{FpLd} & \Rightarrow & \mathit{InstFpLdSpec}\ T_i\ T_o & (*\text{floating}-\text{point load}*) \\
|\ \ \mathit{FpSt} & \Rightarrow & \mathit{InstFpStSpec}\ T_i\ T_o & (*\text{floating}-\text{point store}*)
\end{array}
\end{array}
$$

Definition 6.3 has eight different cases, one for each of the eight different classes of instructions that the microprocessor supports. The definitions of the specifications for each of the eight classes are related to the datapaths of the pipeline and so are not important for our discussion.

When verifying a pipeline using the framework, there are three main goals that we must prove: the stages are correct, the specifications for the framework parameters are met, and the combinational representation implies the specification (Section 4.3.9). For this pipeline all of the stages are unit-delay. We used the standard implementations for unit-delay stages to build the cores and used the unit-delay matching relation. So we know that the stages are implemented correctly.

When using the decision trees, we did not incur any obligations. So, we know that that pipeline conforms to the specifications for the framework parameters. The only remaining goal is to show that the combinational representation implies the specification, which is datapath verification. Because we are not concerned with the datapath for this example, we will leave the proof at this point.

6.4 Dual-Issue Instruction Pipeline

6.4.1 Description

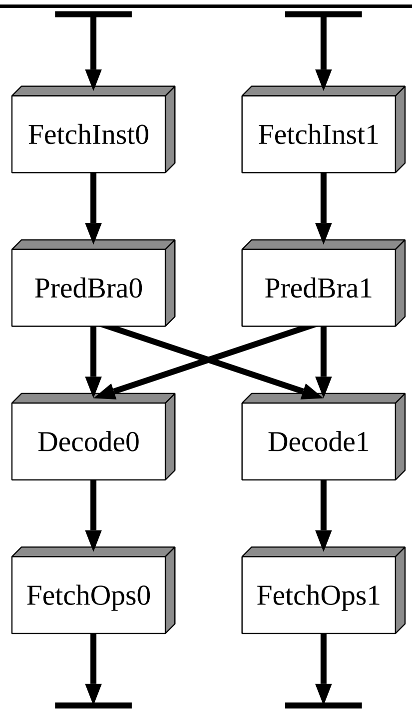


Figure 6.3: Dual-issue instruction pipeline

The primary difference between the single-issue instruction pipeline described in Section 6.3 and the pipeline described in this section is that this pipeline is super-scalar. This pipeline consists of two pipelines from Section 6.3 executing in parallel. The left hand pipeline is named I0 and the right-hand pipeline is I1. Transactions in one pipeline may switch to the other pipeline after the PredBra stage.

The PredBra stage is necessary in the ADK because each of the two halves of the instruction pipeline can only issue four of the eight instruction classes. For example, I1 can issue floating-point operations, but I0 can not. This is due to the connectivity of the pipeline to the register files and execution pipelines (Figure 6.1). Instructions are fetched on two-word aligned blocks: transactions at even addresses go to I0 and transactions at odd addresses

go to I1. If a floating-point transaction is stored at an even address in memory, it will enter the I0 pipeline and then switch to the I1 pipeline after the PredBra stage.
If the transactions in both PredBra stages need to switch pipelines, then the transaction in PredBra1 will transfer to Decode0 and the transaction in PredBra0 will go to Decode1 without any collisions. If both of the transactions wish to use the same Decode stage, then the transaction in PredBra0 will transfer first and the transaction in PredBra1 will stall. The pipeline does not always fetch two instructions at a time. For example, if the last instruction in a procedure is at an even address, then only the I0 pipe will receive a transaction that cycle.
Transactions proceed in pairs from the FetchInst to the PredBra stages and from the Decode to the FetchOps stages. Because of the possible need to split the pair of transactions in the PredBra stage (both transaction need to go to the same side of the pipeline), the second transaction (the one in PredBra1) of a pair may stall while the first (in PredBra0) proceeds. When a transaction enters one of the FetchOps stage it stays within the stage until its operands are available. When its operands are available, it requests to exit the instruction pipeline. Because both transactions in a pair transfer from Decode to FetchOps simultaneously, both of the Decode stages will stall if one of the FetchOps stages is unable to accept an incoming transaction.
A pair of stages that do not contain any transactions are said to contain a *bubble*. The instruction pipeline will *squash bubbles* when there is a pipeline stall. This happens when one pair of stages contains a bubble and has an incoming request. The stalled stages will remain stalled, but the stages behind the bubble proceed and the bubble disappears.
The pipeline handles control hazards by preserving the order of branches and aborting transactions that follow a branch that was mispredicted. Data hazards are resolved by stalling a transaction in the FetchOps stage if its operands are not ready yet.

6.4.2 Instantiations for Framework Parameters

We use the decision trees (Figures 4.1, 4.2, 4.3, and 4.4) to choose instantiations for the four parameters to the framework. Table 6.3 shows the instantiations for the framework parameters.

Protocol The environment can delay accepting requests, so we use a general protocol scheme.

Arbitration The pipeline does not have a tree topology and we are not using a transit protocol scheme, but the topology is a DAG. The topology rules for a static arbitration scheme are not met, because there are multiple entry ports. By forcing the individual FetchInst and PredBra stages to proceed in lockstep, the “no passing” rules for static arbitration are met, and so we can use a static arbitration scheme.

Control The topology of the pipeline is a DAG, so the topological rules for a control scheme are met.

Ordering There are two entry stages, so we need to assign tags to transactions. Because there are two entry and two exit ports, when we identify a transaction entering or leaving the pipeline, we will need to supply both the time and the port. There are several ordering schemes for tags (uniform latency, unique, and in-order) and two common ways of assigning tags to transactions (instruction class or result destination).

By process of elimination, we arrive at a tags-in-order scheme: The pipeline can stall, so we can not use a uniform latency ordering scheme. The input stream can contain adjacent transactions that are in the same instruction class (*e.g.* an addition followed by a subtraction) or that have the same destination register, so we cannot use a unique-tags ordering scheme. All transactions with the same instruction class exit from the same port, and transactions that exit from the same port exit in the order that they enter the pipeline. So by tagging transactions with their instruction class, we can use a tags-in-order ordering scheme.

Table 6.3: Framework instantiations for dual-issue instruction pipeline

Protocol	General
Arbitration	Static
Control	Topological rules
Ordering	Tags-in-order

6.4.3 Stage Instantiations

We are using a general protocol scheme; this means that all of the stages are general stages, except for FetchOps0 and FetchOps1, which are multi-delay stages. A transaction must remain in the FetchOps datapath until its operands are available.

6.4.4 Stage Components

The only stages that have multiple source or destination stages are the PredBra and Decode stages. The PredBra stages have two possible next stages (Decode0 and Decode1) and conversely the Decode stages can receive transactions from either of the PredBra stages. The arbitration scheme is static-prioritization and gives higher priority to requests from PredBra0 than PredBra1. This helps ensure that transactions exit the pipeline in-order. We use the standard implementations (Section 4.3.7) for the interface circuitry for the PredBra and Decode stages and the arbitration circuitry for the Decode stages.

The control circuitry for the PredBra stages needs to select which Decode stage to send transactions on to. This decision is done solely on the basis of the instruction class of the transaction, and with a small effort when encoding the instructions, we can perform this operation simply by looking at one bit in the opcode. For example, if the MSB is a zero, then we send the transaction to the I0 port, otherwise we send it to I1. The hardware needed to do this is trivial to implement.

6.4.5 Combinational Representation and Matching Relation

The combinational representation of the pipeline is shown in Definition 6.4. The representation is a relation over pairs of type *port* $\times$ *transaction* (Section 4.3.10). The first pair $(p_i,\ T_i)$ is an input port and transaction, and the second pair is for the corresponding output port and transaction. If the input port is I0, then the transaction goes through FetchInst0 and PredBra0, otherwise it uses the I1 side. Similarly, the transaction will use one of the two sets of the Decode and FetchOps stages, depending on what the output port of the transaction is.

Definition 6.4: *Instruction pipeline – combinational representation*

$$
\begin{array}{l}
Inst_M\ (p_i,\ T_i)\ (p_o,\ T_o)\ \hat{=} \\
\quad \exists\, T_2\ . \\
\qquad\qquad case\ p_i \\
\qquad\qquad\quad of\ I0\ \Rightarrow\ (\exists\, T_1.\ FetchInst0_M\ \ T_i\ \ T_1\ \&\ PredBra0_M\ \ T_1\ \ T_2) \\
\qquad\qquad\quad \mid\ \ I1\ \Rightarrow\ (\exists\, T_1.\ FetchInst1_M\ \ T_i\ \ T_1\ \&\ PredBra1_M\ \ T_1\ \ T_2) \\
\qquad\ \ \&\ \ case\ (exitPort\ \ T_2) \\
\qquad\qquad\quad of\ I0\ \Rightarrow\ \ (p_o = I0\ \ \& \\
\qquad\qquad\qquad\qquad\quad \exists\, T_3.\ Decode0_M\ \ T_1\ \ T_3\ \ \&\ \ FetchOps0_M\ \ T_3\ \ T_o) \\
\qquad\qquad\quad \mid\ \ I1\ \Rightarrow\ (p_o = I1\ \ \& \\
\qquad\qquad\qquad\qquad\quad \exists\, T_3.\ Decode0_M\ \ T_1\ \ T_3\ \ \&\ \ FetchOps0_M\ \ T_3\ \ T_o)
\end{array}
$$

We can simplify Definition 6.4, because the datapaths on each side of the pipeline are identical. In Definition 6.5 we replace each datapath suffixed with “0” or “1” with the non-suffixed version and separate the calculation of the exit port from the data. In this definition, the functionality of the datapath is independent of the input and output ports. This explains why the input port (p_i) is not used in the body of the definition. The equivalence of these two definitions is captured in Lemma 6.1.

Lemma 6.1: *Equivalence of simplified combinational representation*

$$
\begin{array}{l}
\vdash\ \forall\ p_i,\ p_o,\ T_i,\ T_o\ . \\
\qquad Inst_M Simp\ (p_i,T_i)\ (p_o,T_o)\ \iff\ Inst_M(p_i,T_i)\ (p_o,T_o)
\end{array}
$$

From Table 6.3 we use a tags-in-order matching relation (Definition 4.6), where the tags are the instruction class of the transactions.

Definition 6.5: *Simplified combinational representation for instruction pipeline*

$$
\begin{array}{l}
\mathit{Inst}_M\mathit{Simp}\ (p_i, T_i)\ (p_o, T_o) \hat{=} \\
\quad \exists\, T_1,\ T_2,\ T_3\ . \\
\qquad \mathit{FetchInst}_M\ \ T_i\ T_1\ \ \& \\
\qquad \mathit{PredBra}_M\ \ T_1\ T_2\ \ \& \\
\qquad \mathit{Decode}_M\ \ T_2\ T_3\ \ \& \\
\qquad \mathit{FetchOps}_M\ \ T_3\ T_o\ \ \& \\
\qquad p_o = (\mathit{exitPort}\ T_2)
\end{array}
$$

Because we are using a tagged matching relation (*tagsInOrder*), we need to compute a tag for each transaction. This is in contrast to the non-tagged matching relations in Chapter 5, which matched transactions using request and accept signals and were able to be used directly by the pipeline. For this pipeline, the tag for a transaction is its instruction class. A stream of transactions for a multi-port pipeline is a function from ($\mathit{time}\times\mathit{port}$) to ($\mathit{port}\times\mathit{trans}$). We use the function *mkTag* (Equation 6.1) to compute the tag for a ($\mathit{port}\times\mathit{trans}$) entry in the input or output stream. The function *mkTag* also says whether the entry in the stream represents a transaction. This is done by checking the request and accept signals. If both are high, then a transaction was transferred through the port in that clock cycle. To match two transactions for the pipeline, (Equation 6.2), we apply *mkTag* to each entry in the input and output streams and use the *tagsInOrder* matching relation.

$$\mathit{mkTag}\ (p,\ (T,\ (\mathit{req},\ \mathit{acc}))) \hat{=} ((\mathit{class} \circ \mathit{opcode})\ T,\ (\mathit{req}\ \&\ \mathit{acc})) \tag{6.1}$$

$$\mathit{matchInst}\ I\ O \hat{=} \mathit{tagsInOrder}\ (\mathit{map}\ \mathit{mkTag}\ I)\ (\mathit{map}\ \mathit{mkTag}\ O) \tag{6.2}$$

6.4.6 Verification of Pipeline

The behavioral specification for the instruction pipeline is shown in Definition 6.6. The specification, just like the combinational representation (Definitions 6.4 and 6.5), is a relation over an input port and transaction and an output port and transaction. The specification contains two clauses. The first clause says that the exit port is correct, based on the instruction class of the input transaction. The second clause uses *InstDataSpec* (Definition 6.3) to say that the data in the transaction is correct. The specification of the pipeline is independent of the input port, and so the input port p_i does not appear in the specification.

Definition 6.6: *Specification for instruction pipeline*

$$
\begin{array}{l}
\mathit{Inst}_S\ (p_i,\ T_i)\ (p_o,\ T_o) \hat{=} \\
\quad p_o = \mathit{if}\ (\mathit{class}\ (\mathit{opcode}\ T_i)) \in \{\mathit{FpBr},\ \mathit{IntOp},\ \mathit{IntLd},\ \mathit{IntSt}\ \} \\
\qquad \mathit{then}\ \mathit{I}0\ \mathit{else}\ \mathit{I}1 \\
\quad \&\ \mathit{InstDataSpec}\ T_i\ T_o
\end{array}
$$

The differences between this pipeline and the single-issue pipeline in Section 6.3 are that this is one is dual-issue, the environment can delay accepting requests, and transactions may need to use the last stage for more than one clock cycle. These issues are independent of the specifications for the datapaths of the stages and are only related to the pipelining circuitry.

As with any pipeline, there are three primary goals that we must solve: the stages are correct, the framework specifications are met, and the combinational representation implies the specification.

6.4.6.1 Stages Are Implemented Correctly

All of the stages are in the "general" class of stages, except FetchOps0 and FetchOps1, which are "multi-delay" stages. We used the standard implementations for the request/accept generation hardware and matching relations. If we built the combinational representations of the datapaths by replacing each sequential component with the corresponding combinational component, then these conditions are sufficient to prove that the general stages are modeled correctly.

The FetchOps stages have their own cores that generate request and accept signals, because transactions may remain within the core for multiple clock cycles. The only aspect of the verification of a multi-delay stage that is different from general stages is that we need to prove that every transaction will request to exit the datapath. For the FetchOps stages, this puts an obligation on the environment that the operands for every transaction eventually arrive. Proving that this property holds is done by showing that every operand is already in the register file or is the result of a transaction currently in an execution pipeline. Section 6.5 describes the verification of the floating-point pipeline, which (using the framework) includes showing that every transaction issued to the pipeline is guaranteed to terminate. We could prove similar properties for the other two pipelines that update the register files (the integer and load/store pipelines).

6.4.6.2 Framework Parameters

To show that the specifications for the framework parameters are met, we need to verify the following obligations, which were incurred while using the decision trees (Section 6.4.2):

1. The protocol implementation meets the specification for a general protocol scheme.
2. The no-passing rules for a static-arbitration are met.
3. The control circuitry in the PredBra stages routes transactions to the correct Decode stage.
4. Transactions with the same tag exit in the order that they entered.

We will skip the first obligation because we used standard implementations of the protocol hardware for the stages. The third obligation is solved by verifying the correctness of the simple circuit for choosing how to route transactions, which was described in Section 6.4.4. The second and fourth obligation are the more interesting ones.

To prove that the no-passing rules are met, we need to show that once a transaction in some stage (i) has begun to request a second stage (j), transactions from later in the instruction stream can not request j from a stage that has higher priority than i until the transaction in i has been accepted. The only stages that face contention are the Decode stages. So we need to prove that when a transaction requests a Decode stage, transactions from later in the instruction stream can not request the Decode stage from a higher priority stage until the first transaction has been accepted.

In the process of proving the fourth obligation (that transactions with the same tag exit in-order), we will also prove this property about arbitration and the Decode stages. To prove that transactions with the same tag exit the pipeline in the same order that they entered, we prove that transactions with same tag exit from the same port, and then that transactions enter the Decode stage in the order they enter the pipeline.

Based upon the fact that the tag of a transaction is its instruction class, and using the results from the second obligation, we know that transactions with the same tag enter the same Decode stage and hence exit the pipeline from the same port. In the next paragraph we prove that all transactions that use the same Decode stage enter the decode stage in the order that they enter the pipeline. This is condition is strong enough to also prove that transactions that request a Decode stage will have their request accepted before any transaction later in the stream requests the stage from a higher priority stage.

The arbitration scheme for the Decode stages is static: PredBra0 has priority over PredBra1. There is only contention when both transactions in a pair wish to transfer to the same Decode stage. When this happens, the PredBra0 transaction transfers first and the PredBra1 transaction stalls until the next available clock cycle. While PredBra1 is stalled, PredBra0 does not accept any incoming transactions. This solves the obligation that transactions from later in the stream do not request a Decode stage before earlier requests have been accepted. Thus, we have shown that the specifications for the framework parameters are met.

6.4.6.3 Verification of Functionality

Figure 6.4 shows the beginning of the proof that the combinational representation implies the specification. In the first step we move the combinational representation ($Inst_M$) into the hypothesis list and then use Lemma 6.1 to replace $Inst_M$ with the simplified version $Inst_M Simp$. When we unfold the combinational representation ($Inst_M Simp$) and specification ($Inst_B$), we are left with a sequent that is identical to that of the single-issue instruction pipeline with the added obligation to show that the exit port (p_o) is correct.

All of the complexity introduced by having a dual-issue pipeline is taken care of in the verification of the pipelining circuitry. It does not affect the verification of the datapath, except for the additional subgoal to show that the output port is correct. This subgoal can

$\vdash\ \forall\, TransTop\ TransBot\ .$
$\quad Inst_M\ TransTop\ TransBot \implies Inst_S\ TransTop\ TransBot$

BY `Begin proof`

- $Inst_M\ TransTop\ TransBot$

$\vdash\ Inst_S\ TransTop\ TransBot$

BY `Lemma 6.1`
(* Substitute $Inst_M Simp$ for $Inst_M$ *)

- $Inst_M Simp\ TransTop\ TransBot$

$\vdash\ Inst_S\ TransTop\ TransBot$

BY `Unfold Definitions 6.5, 6.6` (* $Inst_M Simp$, $Inst_S$ *)

- $FetchInst_M\ T_i\ T_1$
- $PredBra_M\ T_1\ T_2$
- $Decode_M\ T_2\ T_3$
- $FetchOps_M\ T_3\ T_o$
- $p_o = (exitPort\ T_3)$

$\vdash\ InstDataSpec\ T_i\ T_o$

$\vdash\ p_o = (if\,(class\,(opcode\ T_i)) \in \{FpBr,\ IntOp,\ IntLd,\ IntSt\}$
$\qquad then\ `I0'\ else\ `I1')$

Figure 6.4: Verification of functionality of instruction pipeline

be solved by examining the function *exitPort*, and verifying that it correctly returns the index for either I0 or I1.

6.5 Floating Point Pipeline

6.5.1 Description

The floating-point pipeline in the ADK consists of five main stages (AddRecode, MultShift, AddNorm, AddRound, and WriteBack) plus a zero-delay routing stage (Router) at the top and the Divide stage. Most transactions go through the router and each of the main stages once. The two exceptions are multiplies, which use the MultShift stage twice, and divides, which use the Divide stage a data-dependent number of times and skip the MultShift stage. The Router stage is purely combinational logic and is used to send input transactions to either the Divide or AddRecode stage.

The VAX 8600 model 200 (NVAX) pipeline supports the standard four arithmetic operations (addition, subtraction, multiplication, and division) and some additional operations for many different data formats and includes operations to convert between the formats. In our simplified design, we support the four basic operations for one single format. The inclusion of the additional formats and conversion operations only affects the number of operations that need to be verified on the datapath, it does not affect the use of the framework.

In a continuous stream of transactions that do not contain any divide or multiply transactions, there will not be any collisions and a new transaction can be issued each clock cycle. The divide stage can generate between one and three bits of a result quotient each cycle. Double precision floating-point numbers contain fifty-four bits for the mantissa, resulting in a latency of between twenty-two and fifty-eight cycles for divide transactions. Only one divide transaction can be within the divide stage at a time.

Following the design of the NVAX, this pipeline does not accept transactions into the pipeline if they will collide with a transaction already in the pipeline. We use a circuit in the entry stage of the pipeline to predict what stages will be available in the future. There are three paths through the pipeline, shown in the reservation tables in Figures 6.6 – 6.8.

6.5.2 Framework Instantiation and Verification

We use the decision trees (Figures 4.1, 4.2, 4.3, and 4.4) to choose instantiations for the four parameters to the framework. Table 6.4 shows the instantiations for the framework parameters.

Protocol The bottom environment cannot delay requests and the top environment is not constrained to submit a new transaction every cycle. So we incur the obligation that stages must accept a request if they are not busy. The topology of the pipeline is not a tree. The top environment does not prevent collisions, but the entry stage does, so

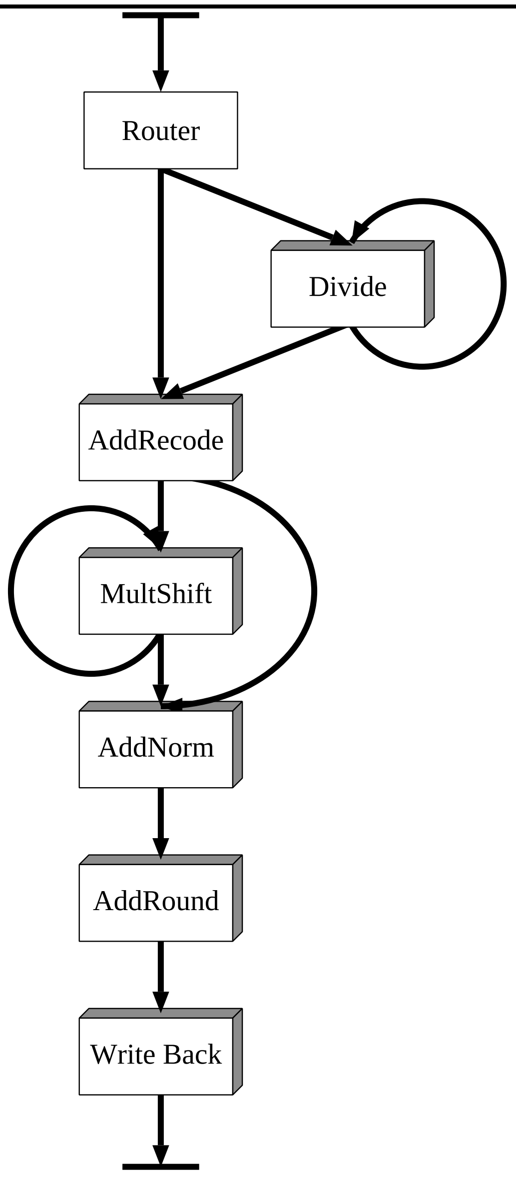


Figure 6.5: Block diagram of NVAX style floating-point pipeline

AddRecode	•					
MultShift		•				
AddNorm			•			
AddRound				•		
WriteBack					•	

Figure 6.6: Normal

AddRecode	×					
MultShift		×	×			
AddNorm				×		
AddRound					×	
WriteBack						×

Figure 6.7: Multiply

AddRecode	÷					
MultShift						
AddNorm		÷				
AddRound			÷			
WriteBack				÷		

Figure 6.8: Divide
Reservation tables for floating-point pipeline

we incur the obligation to prove that the entry stage prevents collisions and we are able to use a transit protocol scheme.

Arbitration The stages that have multiple sources are: the Divide stage, which can receive inputs from either itself or the entry port to the pipeline; the AddRecode stage, whose inputs are connected to the entry port and the Divide stage; the MultShift stage, which can self-loop and is connected to the AddRecode stage; and the AddNorm stage, which can receive transactions from the AddRecode stage or the MultShift stage.

We do not have a tree topology, but use a transit protocol scheme. This means that we can use an exclusive arbitration scheme. We must ensure that stages accept whichever request is active.

Control The topology of the pipeline is not a directed acyclic graph, so we have the obligation to prove that loops in the pipeline will terminate. The two loops are the multiply, which uses the MultShift stage twice, and the divide, which uses the Divide stage a data-dependent number of times.

Ordering There is only one entry stage and transactions exit each stage in the order they enter the stage. Not all transactions take the same path through the pipeline, but there is only one exit stage. Transactions using the divide stage have a longer latency to the AddRecode stage than those that skip the Divide stage, so there is not the same latency to joins. Because of the long latency through the Divide stage, transactions do not enter AddRecode in the same order that they enter the pipeline. So, we must assign tags to transactions.

As with most pipelined microprocessors, when the issue logic in the instruction pipeline issues a transaction to an execution pipeline, it locks the destination register of the transaction to prevent other transactions with that same destination register from being issued to the pipeline. This prevents write-after-write data hazards. It also provides a method for ordering transactions, because it ensures that only one transaction with a given destination register is active in the pipeline at any point in time. By defining the tag of a transaction to be the destination register of the transaction, we can use a tags-unique ordering scheme. So, we must prove that there is at most one transaction with any given tag inside the pipeline at a time.

6.5.3 Stage Instantiations

The pipeline uses a transit protocol scheme, so all of the stages are unit-delay stages, except for the Router stage, which is a zero-delay stage.

Table 6.4: Framework instantiations for NVAX style floating-point pipeline

Protocol	Transit
Arbitration	Exclusive
Control	Loops terminate
Ordering	Tags unique

Table 6.5: Stage classes for floating-point pipeline

Router	Zero-delay
Divide	Unit-delay
AddRecode	Unit-delay
MultShift	Unit-delay
AddNorm	Unit-delay
AddRound	Unit-delay
WriteBack	Unit-delay

6.5.4 Request and Accept Generation

In a transit protocol scheme, we use request signals between stages so that the stages know when they are active and where to receive incoming transactions from. Accept virtual signals between stages are always true. We use a pair of request and accept signals between the top environment and the floating-point pipeline.

In this pipeline we use a protocol circuit that only accepts transactions from the environment when it can guarantee that they can traverse through the pipeline without encountering any transactions. Our protocol circuit is a slightly modified version of Davidson's shift-register circuit [DSTP75]. To simplify the protocol circuit, we put it in the AddRecode stage rather than in the Router stage. This is done so that we can easily handle the variable latency of the Divide stage. When a divide transaction is accepted into the pipeline, we can not say exactly when it will exit the divide stage. When the Router stage receives a request from the environment, it forwards it to either the Divide stage or the AddRecode stage, where their protocol circuitry decides whether to accept the request or not. The AddRecode stage uses its state machine to find out whether the transaction will collide or not and sends out an accept if the transaction will not collide with any transaction already in the pipeline. The Divide stage accepts the request if it is idle, otherwise it denies it. The Router forwards the response to the environment. From this description, it is clear that the Router stage consists of nothing more than multiplexers, de-multiplexers, and a small block of logic.

6.5.5 Interface, Arbitration, and Control Circuitry

In this pipeline the interface circuitry uses the standard implementations of multiplexers and de-multiplexers and the arbitration units are priority encoders. We need to design the control circuits for the Divide and MultShift stages. For the Divide stage, we examine the value of the counter to detect when the loop is finished. All of the other paths through the pipeline are uniquely determined by the opcode of the transaction. If each stage were used at most once in any path, then we could determine the operation that each stage is to perform and the next stage to go to based solely on the opcode, but the MultShift stage is used twice by multiplies. So, we include an extra bit in the data-stationary control for the MultShift that determines whether it is a transaction's first or second time through the stage.

6.5.6 Verification of Floating-Point Pipeline

We must verify that the stages are implemented correctly, the framework parameters are instantiated correctly, and the combinational representation of the pipeline implies the specification. The calculation of the combinational representations of the stages follow the standard methods and do not warrant any special attention.

6.5.6.1 Verification of Framework Parameter Instantiations

The obligations that we incurred in walking through the decision trees are:

1. If stages are not busy, they must accept a request.
2. Whenever a transaction is accepted into the pipeline, it is guaranteed not to collide.
3. Stages must accept whatever request is active.
4. All loops in paths through the pipeline terminate
5. There is only one transaction with a given destination register active in the pipeline at a time.

We use standard protocol and arbitration circuitry (priority encoder) for each stage that has multiple sources, this satisfies the first and third obligation. The second and fourth obligations are the most involved to check, so we briefly postpone their discussion. For the fifth obligation, the environment is responsible for not issuing a transaction if there is a transaction with the same destination register already in the pipeline.

For the fourth obligation, we need to prove that all loops terminate. The two loops in the pipeline are multiplies, which use the MultShift stage twice and divides, which use the Divide stage a data-dependent number of times. We use a single bit to detect whether it is a transaction's first or second time through the MultShift stage. On the first time through, the value of the bit is toggled and the transaction is sent back to the MultShift stage. On the second time through the transaction is sent on to the AddNorm stage. For divide transactions, we begin with a finite upper bound (fifty-four) on the loop counter and decrement the counter by one, two, or three (depending how many bits of the quotient we calculate that cycle) each iteration until we reach zero.

When a transaction requests to enter the pipeline, the Router stage sends the request to either the Divide stage or the AddRecode stage. If it is a divide transaction, the Divide stage will accept it if it is not currently busy, otherwise it will deny the request. The AddRecode stage uses its protocol circuit to determine if the transaction will collide with any transaction already in the pipeline.
The method described in literature for designing the protocol circuit is sufficient to prevent collisions within the main stages of the pipeline. This pipeline is somewhat unique though, because we need to ensure that a transaction in the Divide stage will not stall when it wants to transfer to AddRecode. Because of the variable latency of the Divide stage, we chose not to include the Divide stage in the protocol circuitry. The Divide stage sends a request signal to the protocol circuitry two cycles before the transaction is ready to transfer. This allows the protocol circuitry to deny any requests from incoming transactions that would conflict with the divide transaction when it enters the main stages.

6.5.6.2 Verification of Functionality

Definition 6.7: *Combinational representation of ADK floating-point pipeline*

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

Each of the four operations we describe has its own path through the pipeline, as shown in the combinational representation ($Float_M$, Definition 6.7). For conciseness we only include the paths for the addition and division operations. $Float_M$ existentially quantifies over the internal transactions in the pipeline and does 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 decrement the loop counter a data-dependent amount each iteration, until it reaches zero. The amount is calculated by applying the function *decVal* to the transaction in the stage. To verify the divide operation, we would use a loop invariant style of proof. The other paths through the pipeline are relatively straightforward.

Definition 6.8: *Specification of ADK floating-point pipeline*

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

The specification for the pipeline (Definition 6.8) does a case split on the opcode of the input transaction and then checks that the result ($\mathit{getResult}\ T_o$) is the correct result for the input operands ($\mathit{getIOps}\ T_i$).

The two sources of complexity in this pipeline are the protocol circuitry and the data-dependent loop for the Divide stage. As demonstrated here, we were able to isolate these concerns from each other and from the verification of the datapath. In verifying the functionality of the pipeline, the loop for the Divide stage can be verified using any of the standard loop verification techniques from program verification.

6.6 Framework Parameters for Complete Microprocessor

In this section we combine the internal pipelines of the ADK together to create the complete microprocessor. Figure 6.9 shows the top level pipeline of the ADK and the five internal pipelines.

6.6.1 Framework Instantiations

Protocol There are six exit ports:

1. integer pipeline to integer register file
2. load/store pipeline to integer register file
3. load/store pipeline to data cache
4. load/store pipeline to floating-point register file
5. branch unit to program counter
6. floating-point pipeline to floating-point register file

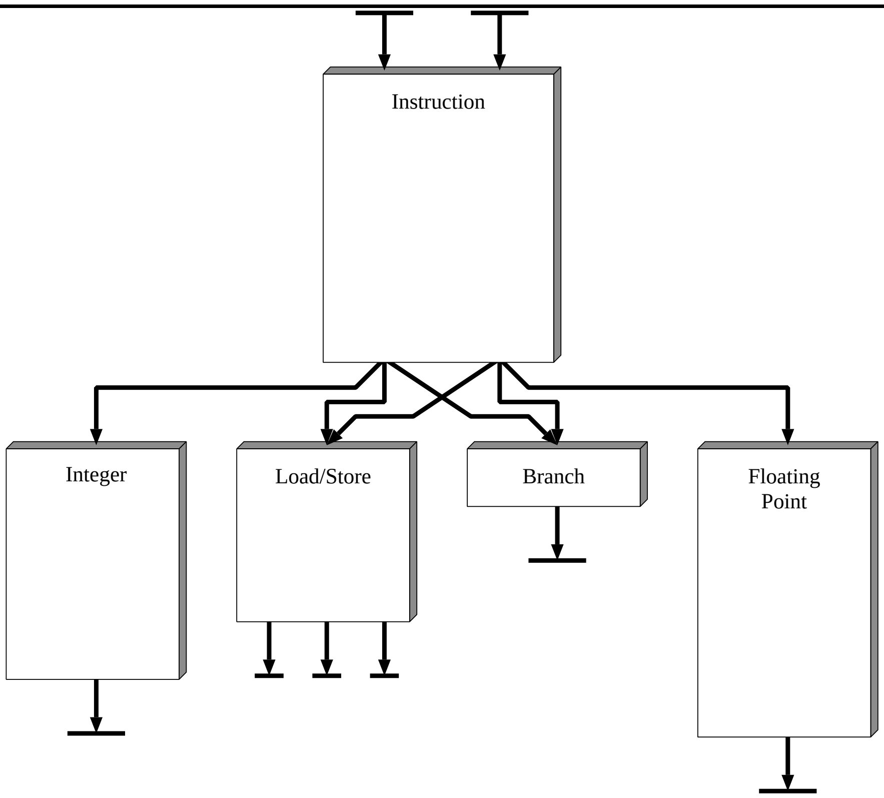


Figure 6.9: Top level of ADK microprocessor

The only one of these ports where the environment can delay requests is from the load-store pipeline to the data-cache. If the cache is busy it cannot accept a new request right away. The accept virtual signals for the program counter and register files are always true. Because one of the exit ports can delay requests, we have to use a general protocol scheme.

Arbitration We do not have a tree topology, because there is contention for the Branch and Load/Store pipelines. We are not using a transit protocol scheme, but the topology is a directed acyclic graph. Because there are two entry ports, the pipeline does not conform to the topology rules for a static-prioritization scheme.

We now check if the pipeline meets the no-passing rules for a static arbitration scheme. From the arbitration scheme for the instruction pipeline, we know that transactions will stall in the Decode stages until both transactions in the FetchOps stages have transferred to the execution pipes. This means that when a transaction in the instruction pipeline requests an execution pipeline, no transaction from later in the instruction stream will subsequently request the execution pipeline until the first request has been accepted. The only contention we will have is when both transactions in a pair wish to use either the Branch unit or the Load/Store pipeline.

For the purposes of arbitration, it does not matter which transaction we select first. The one that is selected will enter the contested pipeline and no other transaction will request the pipeline until the second transaction has been accepted.

For control and data hazards though, we need the pipeline to appear to execute transactions in-order. If both transactions in a pair are branches, where one is a floating-point branch and one an integer branch, the floating-point branch will always exit from the I0 side of the instruction pipeline and the integer branch will exit from the I1 side of the pipeline, regardless of what order they appeared in the input stream. Similarly, if we have an integer store and a floating-point store, the integer store will always exit from the I0 side and the floating-point store will exit from the I1 side. To execute programs correctly, we need to execute the branches in the same order that they appeared in the instruction stream. If both stores write to the same location in memory, then we must execute them in-order. So, the Branch and Load/Store pipelines need to accept transactions in the order that they were issued to the pipeline, where the I0 entry port is before the I1 port.

We can do this by detecting when transactions in a pair swap sides in the instruction pipeline. If transactions do not swap, then the transaction from the I0 port should be accepted first. If the transactions swap sides, then the Branch Unit and Load/Store pipeline need to accept the transaction in the I1 port first. This means that we can not assign a static priority to the exit ports of the instruction pipeline and thus we can not use a static-prioritization scheme. The control circuitry of the pipelines does not prevent contention, so we end up with a general arbitration scheme.

Control The topology of the pipeline is a directed acyclic graph, and so the pipeline conforms to the topological rules for a control scheme.

Ordering There may be more than transaction per stage, so we assign tags to transactions. In the instruction pipeline we tagged transactions by their opcode class and in the floating-point pipeline we tagged transactions by their destination. The other execution pipelines were not described in this thesis, but for the Integer and Load/Store pipelines we tag transactions with their destination. For the Branch pipeline, there is at most transaction at a time, and so we can trivially tag transactions with their destination, which is the program counter.

All transactions with the same opcode class exit the instruction pipeline in-order and go to the same execution pipeline. All transactions with the same destination exit the execution pipelines in-order. So, we tag transactions with their opcode-class and destination, and use a tags-in-order ordering scheme for the ADK.

Table 6.6: Framework instantiations for top level of ADK microprocessor

Protocol	General
Arbitration	Static-priority
Control	DAG
Ordering	Tags-in-order

6.6.2 Segments

All of the pipelines except for the Branch unit generate request and accept signals internally. The branch unit contains at most one transaction, transactions spend exactly one cycle within the unit, and requests from the unit to the program counter are always accepted immediately. Thus, the branch unit is always able to accept an incoming transaction.

We now proceed with the design of the interface, arbitration, and control functional units. The two pipelines that can face contention are the Load/Store pipeline and the Branch unit. The arbitration hardware for these pipelines is identical. It selects the I0 or I1 port of the instruction pipeline depending upon if there is contention and if the transactions swapped in the instruction pipeline.

The integer and floating-point pipelines have only one source, and so do not require any arbitration or interface hardware. The interface hardware for the input ports of the Branch unit and Load/Store pipeline and the two output ports of the instruction pipeline are the standard implementations. The Load/Store pipeline generates an accept signal for incoming transactions. Its interface hardware sends this accept signal to the higher priority request and denies the lower priority request. Thus if the pipeline can accept an incoming transaction, the high priority request is accepted, but otherwise is denied. The Branch unit is always able to accept an incoming request, so the interface unit always accepts the high priority request and denies the low priority request.

6.6.3 Verification

6.6.3.1 Segments Modeled Correctly

In this chapter we described the verification of the instruction and floating-point pipelines. For both of these pipelines we proved that the combinational representation models the implementation and matching relation. We could prove similar results for the Integer, Load/Store, and Branch pipelines, all of which are simpler than the floating-point and instruction pipelines.

6.6.3.2 Framework Specifications

The obligations that we incurred while going through the decision trees are:

1. The specification for a general protocol scheme is met.
2. If the internal pipelines eventually accept every request, then the arbitration circuitry for the Load/Store and Branch pipelines must eventually select every request as the highest priority request.
3. The control circuitry for the exit ports of the instruction pipeline routes transactions to the correct execution pipeline.
4. Transactions with the same exit port and destination exit the pipeline in-order.

For the first obligation, we have to show that the specification for a general protocol scheme is met. All of the pipelines generate request and accept signals internally, except for the Branch unit, which is always able to accept an incoming transaction. By using the framework to design and verify the pipelines (assuming that we did this for the Integer, Load/Store, and Branch pipelines also), we know that these signals are correct for a general protocol scheme. We used standard implementations of the interface hardware to connect the stages. This is sufficient to show that we conform to the specification for a general protocol scheme.

For the second obligation, we have to show that the arbitration circuitry meets the specification for a general arbitration scheme. We outlined the argument for correctness in the discussion of the arbitration decision tree. To summarize: transactions proceed in pairs through the instruction pipeline. The only source of contention is if both transactions in a pair are destined for different sides of the instruction pipeline, but the same execution pipeline. In this case they will simultaneously request the same execution pipeline. We give higher priority to the transaction that appeared earlier in the instruction stream. The instruction pipeline will not issue a request from a new pair of transactions until both transactions in the previous pair have been accepted. Thus the second transaction is guaranteed to eventually be accepted.

In the third obligation we check that the control circuitry for routing transactions from the instruction pipeline to the execution pipelines behaves correctly. As usual, this is very simple circuitry that is straightforward to verify, so we will not discuss it here.
To verify the ordering scheme, we need to show that transactions with the same exit port and destination exit the pipeline in-order. From the correctness results of the instruction pipeline, we know that transactions with the same class exit the instruction pipeline in-order. The instruction class of a transaction determines its exit port in the microprocessor. There is a one-to-one correspondence between instruction classes and exit ports except for the port to the data-cache (integer and floating-point stores) and branches (integer and floating-point). For the arbitration scheme, to handle control and data hazards, we chose to accept load/store and branch transactions to the execution pipelines in the order that they appeared in the input stream. All of the execution pipelines execute transactions with the same destination in-order. This is sufficient to show that transactions with the same exit port and destination finish in-order.

6.6.3.3 Functionality

The specification for the complete ADK is shown in Definition 6.9. The specification does a case split on the class of the opcode for the input transaction. The specification is independent of the input port (p_i). For each of the eight classes of instructions, there is a separate specification for the transaction. We also check that the output port is correct.

Definition 6.9: *Specification of ADK*

$$
\begin{array}{l}
ADK_S\ (p_i,\ T_i)\ (p_o,\ T_o)\ \hat{=} \\
\quad case\ ((class \circ opcode)\ T_i) \\
\qquad \begin{array}{lll}
of\ IntBr & \Rightarrow & IntBrSpec\ T_i\ T_o\ \ \&\ \ p_o = \text{`}PC\text{'} \\
\mid\ IntOp & \Rightarrow & IntOpSpec\ T_i\ T_o\ \ \&\ \ p_o = \text{`}intRF_0\text{'} \\
\mid\ IntLd & \Rightarrow & IntLdSpec\ T_i\ T_o\ \ \&\ \ p_o = \text{`}intRF_1\text{'} \\
\mid\ IntSt & \Rightarrow & IntStSpec\ T_i\ T_o\ \ \&\ \ p_o = \text{`}dCache\text{'} \\
\mid\ FpBr & \Rightarrow & FpBrSpec\ T_i\ T_o\ \ \&\ \ p_o = \text{`}PC\text{'} \\
\mid\ FpOp & \Rightarrow & FpOpSpec\ T_i\ T_o\ \ \&\ \ p_o = \text{`}fpRF_0\text{'} \\
\mid\ FpLd & \Rightarrow & FpLdSpec\ T_i\ T_o\ \ \&\ \ p_o = \text{`}fpRF_1\text{'} \\
\mid\ FpSt & \Rightarrow & FpStSpec\ T_i\ T_o\ \ \&\ \ p_o = \text{`}dCache\text{'}
\end{array}
\end{array}
$$

Definition 6.10 is the combinational representation of the ADK. It uses the combinational representations of the internal pipelines and routes transactions from the instruction pipeline to the appropriate execution pipeline. The correctness of the choice of the execution pipeline is captured in Lemma 6.2.
The beginning of the proof that the combinational representation implies the specification is shown in Figures 6.10 and 6.11. We begin by unfolding the combinational representation and specification and then use Theorems 6.1 – 6.5 to replace the combinational represen-

Definition 6.10: *Combinational representation of ADK*

$$
\begin{array}{l}
ADK_M\ (p_i,\ T_i)\ (p_o,\ T_o)\ \hat{=} \\
\quad \exists (p_x,\ T_x)\ . \\
\qquad inst_M\ (p_i,\ T_i)\ (p_x,\ T_x)\ \& \\
\qquad case\ (execPipe\ T_x) \\
\qquad\quad \begin{array}{lll}
of\ \text{`}Integer\text{'} & \Rightarrow int_M\ T_x\ T_o & \&\ p_o = \text{`}intRF_0\text{'} \\
\mid\ \text{`}LoadStore\text{'} & \Rightarrow ldSt_M\ T_x\ (p_o,\ T_o) & \\
\mid\ \text{`}Branch\text{'} & \Rightarrow bra_M\ T_x\ T_o & \&\ p_o = \text{`}PC\text{'} \\
\mid\ \text{`}Float\text{'} & \Rightarrow fp_M\ T_x\ T_o & \&\ p_o = \text{`}fpRF_0\text{'}
\end{array}
\end{array}
$$

Lemma 6.2: *Execute pipeline*

$$
\begin{array}{l}
\vdash \forall T \\
\quad \begin{array}{ll}
(execPipe\ T = Integer) & \Longleftrightarrow\ ((class \circ opcode)\ T = IntOp)\ \& \\
(execPipe\ T = LoadStore) & \Longleftrightarrow \\
\qquad ((class \circ opcode)\ T \in \{IntLd,\ IntSt,\ FpLd,\ FpSt\})\ \& & \\
(execPipe\ T = Branch) & \Longleftrightarrow\ ((class \circ opcode)\ T \in \{IntBr,\ FpBr\})\ \& \\
(execPipe\ T = Float) & \Longleftrightarrow\ ((class \circ opcode)\ T = FpOp)
\end{array}
\end{array}
$$

tations of the internal pipelines (Instruction, Integer, Branch, Load/Store and Float) with their specifications.

Theorem 6.1: *Functionality of instruction pipeline*

$$\vdash \forall\ T_i,\ T_o\ .\ Inst_M\ T_i\ T_o \implies InstSpecData\ T_i\ T_o$$

Using Lemma 6.2, we know that the destination of the transaction exiting the instruction pipeline (T_x) is correct with respect to its instruction class. Using this result, we change the case split in the hypothesis for the execute pipeline to test the instruction class of T_x rather than its destination port. Both the hypotheses and the conclusion are now stated in terms of case splits on instruction classes. We do a case split on the instruction class of the input transaction T_i, propagate correctness results through to T_x, simplify the hypotheses, and are left with eight subgoals, one for each class of instruction. Following our practice of describing only the pipelining logic for the complex pipelines, we will end our discussion of the proof at this point.

6.7 Conclusion

In this chapter we gave an overview of the design and verification of the pipelining circuitry for a fictional super-scalar microprocessor. We showed how the framework can be used to

Theorem 6.2: *Functionality of integer pipeline*
$\vdash \forall\ T_i,\ T_o\ .\ Integer_M\ T_i\ T_o \implies Integer_S\ T_i\ T_o$

Theorem 6.3: *Functionality of load/store pipeline*
$\vdash \forall\ T_i,\ p_o,\ T_o\ .\ LoadStore_M\ T_i\ (p_o,\ T_o) \implies LoadStore_S\ T_i\ (p_o,\ T_o)$

reduce the verification of the datapath of a super-scalar pipeline to that of a single-issue pipeline. The description of the floating-point pipeline demonstrated the verification of a relatively complex pipeline with out-of-order termination and paths that looped through stages multiple times. There are three additional internal pipelines for the microprocessor: integer, load/store, and branch. The pipelining logic for these three pipelines are all simpler than the two pipelines described here. We then combined the internal pipelines together to create the complete microprocessor. In the verification of the functionality of the datapath for the microprocessor, we were able to easily split the proof into eight independent cases, one for each class of instruction. This separation breaks the proof up into manageable size pieces. The individual pieces did not contain any vestiges of the pipelined or super-scalar nature of the implementation of the microprocessor. Once the proof of a pipeline has been simplified to remove complications related to pipelining, standard verification techniques can be applied to finish off the proof. For complex designs, such as the microprocessor described here, these proofs are still quite complex, but if the verification of the pipelining logic had to be tackled simultaneously with the verification of the datapath, the task would be almost infeasible for a realistic super-scalar microprocessor.

Theorem 6.4: *Functionality of branch unit*
$\vdash \forall\ T_i,\ T_o\ .\ Branch_M\ T_i\ T_o \implies Branch_S\ T_i\ T_o$

Theorem 6.5: *Functionality of floating-point pipeline*
$\vdash \forall\ T_i,\ T_o\ .\ Float_M\ T_i\ T_o \implies Float_S\ T_i\ T_o$

$\vdash\ \ \forall p_i,\ T_i,\ p_o,\ T_o\ .$
$\quad ADK_M\ (p_i,\ T_i)\ (p_o,\ T_o) \implies ADK_S\ (p_i,\ T_i)\ (p_o,\ T_o)$

BY `Unfold Definition 6.10 (*` ADK_M `*)`

- $Inst_M\ (p_i,\ T_i)\ (p_x,\ T_x)$
- $case\ (execPipe\ T_x)$
  $of\ \text{`}Integer\text{'} \Rightarrow Integer_M\ T_x\ T_o\ \&\ p_o = \text{`}intRF_0\text{'}$
  $\mid\ \text{`}LoadStore\text{'} \Rightarrow LoadStore_M\ T_x\ (p_o,\ T_o)$
  $\mid\ \text{`}Branch\text{'} \Rightarrow Branch_M\ T_x\ T_o\ \&\ p_o = \text{`}PC\text{'}$
  $\mid\ \text{`}Float\text{'} \Rightarrow Float_M\ T_x\ T_o\ \&\ p_o = \text{`}fpRF_0\text{'}$

$\vdash\ \ ADK_S\ (p_i,\ T_i)\ (p_o,\ T_o)$

BY `Theorems 6.1, 6.2, 6.3, 6.4, 6.5`
(* functionality for instruction, integer, branch, load/store *)
(* and floating-point pipelines *)
`THEN Unfold Definition 6.3 (*` $InstSpecData$ `*)`

- $case\ ((class \circ opcode)\ T_i)$
  $of\ IntBr \Rightarrow InstIntBrSpec\ T_i\ T_x$
  $\mid\ IntOp \Rightarrow InstIntOpSpec\ T_i\ T_x$
  $\mid\ IntLd \Rightarrow InstIntLdSpec\ T_i\ T_x$
  $\mid\ IntSt \Rightarrow InstIntStSpec\ T_i\ T_x$
  $\mid\ FpBr \Rightarrow InstFpBrSpec\ T_i\ T_x$
  $\mid\ FpOp \Rightarrow InstFpOpSpec\ T_i\ T_x$
  $\mid\ FpLd \Rightarrow InstFpLdSpec\ T_i\ T_x$
  $\mid\ FpSt \Rightarrow InstFpStSpec\ T_i\ T_x$
- $case\ (execPipe\ T_x)$
  $of\ \text{`}Integer\text{'} \Rightarrow Integer_S\ T_x\ T_o\ \&\ p_o = \text{`}intRF_0\text{'}$
  $\mid\ \text{`}LoadStore\text{'} \Rightarrow LoadStore_S\ T_x\ (p_o,\ T_o)$
  $\mid\ \text{`}Branch\text{'} \Rightarrow Branch_S\ T_x\ T_o\ \&\ p_o = \text{`}PC\text{'}$
  $\mid\ \text{`}Float\text{'} \Rightarrow Float_S\ T_x\ T_o\ \&\ p_o = \text{`}fpRF_0\text{'}$

$\vdash\ \ ADK_S\ (p_i,\ T_i)\ (p_o,\ T_o)$

Figure 6.10: Functionality proof for ADK – 1

BY `Lemma 6.2 (*` correctness of choice of execution pipelines `*)`
`THEN case split on instruction class of` T_i

- $\mathit{InstIntBrSpec}\ T_i\ T_x$
- $\mathit{Branch}_S\ T_x\ T_o\ \&\ p_o = \text{`}\mathit{PC}\text{'}$

$\vdash \mathit{IntBrSpec}\ T_i\ T_o\ \ \&\ \ p_o = \text{`}\mathit{PC}\text{'}$

- $\mathit{InstIntOpSpec}\ T_i\ T_x$
- $\mathit{Integer}_S\ T_x\ T_o\ \&\ p_o = \text{`}\mathit{intRF}_0\text{'}$

$\vdash \mathit{IntOpSpec}\ T_i\ T_o\ \ \&\ \ p_o = \text{`}\mathit{intRF}_0\text{'}$

- $\mathit{InstIntLdSpec}\ T_i\ T_x$
- $\mathit{LoadStore}_S\ T_x\ (p_o,\ T_o)$

$\vdash \mathit{IntLdSpec}\ T_i\ T_o\ \ \&\ \ p_o = \text{`}\mathit{intRF}_1\text{'}$

- $\mathit{InstIntStSpec}\ T_i\ T_x$
- $\mathit{LoadStore}_S\ T_x\ (p_o,\ T_o)$

$\vdash \mathit{IntStSpec}\ T_i\ T_o\ \ \&\ \ p_o = \text{`}\mathit{dCache}\text{'}$

- $\mathit{InstFpBrSpec}\ T_i\ T_x$
- $\mathit{Branch}_S\ T_x\ T_o\ \&\ p_o = \text{`}\mathit{PC}\text{'}$

$\vdash \mathit{FpBrSpec}\ T_i\ T_o\ \ \&\ \ p_o = \text{`}\mathit{PC}\text{'}$

- $\mathit{InstFpOpSpec}\ T_i\ T_x$
- $\mathit{Float}_S\ T_x\ T_o\ \&\ p_o = \text{`}\mathit{fpRF}_0\text{'}$

$\vdash \mathit{FpOpSpec}\ T_i\ T_o\ \ \&\ \ p_o = \text{`}\mathit{fpRF}_0\text{'}$

- $\mathit{InstFpLdSpec}\ T_i\ T_x$
- $\mathit{LoadStore}_S\ T_x\ (p_o,\ T_o)$

$\vdash \mathit{FpLdSpec}\ T_i\ T_o\ \ \&\ \ p_o = \text{`}\mathit{fpRF}_1\text{'}$

- $\mathit{InstFpStSpec}\ T_i\ T_x$
- $\mathit{LoadStore}_S\ T_x\ (p_o,\ T_o)$

$\vdash \mathit{FpStSpec}\ T_i\ T_o\ \ \&\ \ p_o = \text{`}\mathit{dCache}\text{'}$

Figure 6.11: Functionality proof for ADK – 2

Chapter 7

Correctness and Formal Verification of Pipelines

7.1 Introduction

This chapter and the next (Chapter 8) give the formal foundation of the framework. In this chapter we define what it means for a pipeline to be free of structural conflicts* and satisfy a functional specification. We then describe how to formally verify a pipeline using the framework. The informal design and verification of pipelines was outlined in Chapter 4 and demonstrated, using the *diffAdd* pipeline, in Chapter 5.

This section includes a discussion of how we model pipelines and an introduction to several definitions and lemmas that we use in temporal reasoning. Section 7.2 contains a formal description of what it means for a pipeline to be correct. Section 7.3 describes how we verify pipelines using the framework. Verifying a pipeline requires finding the matching relation and combinational representation and proving that the pipeline meets the specifications of the framework parameters. A variety of commonly used matching relations were described in Section 3.2.4 and the decision tree for finding the matching relation for a pipeline was presented in Section 4.2.4. Section 7.4 describes how to calculate the combinational representation of a pipeline and includes the calculation of the combinational representation of the *diffAdd* pipeline. The framework is based upon the notion of traversing a path through a pipeline and the specification for ordering schemes says that the matching relation for a pipeline correctly identifies when a transaction traverses the pipeline. Section 7.5 introduces our definition of traversing a pipeline and demonstrates this concept with the *diffAdd* pipeline. Section 7.6 presents the formal specifications of the framework parameters.

We often reason about hierarchical pipelines, where the datapaths for the stages may themselves be pipelines. Formally, 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

*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.

lowest level of the structural hierarchy). In this chapter and in Chapter 8, we will use segments except in cases where we specifically mean stages.

7.1.1 Modeling Pipelines

To create a framework that allows us to reason about pipelines in general and then apply the results to specific pipelines, we defined a model of a generic pipeline that is composed of generic segments. The verification of the framework reasons about the model pipeline. This was done by defining a set of virtual functional units and signals in a generic segment and then deriving 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 the control circuitry to be constants that always produce the same output. Chapter 5 describes the design and verification of the *diffAdd* pipeline; in that chapter Section 5.3.2 shows how we defined these functional units for *diffAdd*.

Throughout the rest of this section we give an overview of Figure 7.1, which shows the virtual functional units for a segment and the signals connecting the functional units. Section 7.6 contains the formal specifications of the virtual functional units. Because we are dealing with a single segment in this section, as a notational abbreviation we capitalize the names of the functional units and signals and do not parameterize them by a segment identifier. For example, in this section we refer to *MkInst* and *InstP* rather than *mkInst s* and *instP s*.

The generic segment in Figure 7.1 contains a core, shown in the center, and several peripheral units. The core performs an operation on the data of the transaction. The interface units at the top and bottom connect the segment to the other segments in the pipeline. The arbitration unit chooses which request has the highest priority. The segment accepts a request from segment s if s has the highest priority request and the core can accept an incoming transaction. The control units (*MkInst* and *MkNext*) determine the current instruction to perform and the next segment and operation respectively.

The arbitration unit and interface units comprise the pipelining circuitry. The *mkInst* and *mkNext* units are the control circuitry and the core is the datapath. The combinational representation of a segment reflects the datapath and control circuitry, but not the pipelining circuitry. This ability to separate the pipelining aspects of behavior from the combinational representation is one of the key concepts in the framework. The combinational representation is constructed from the combinational representation of the core (*comb*) and the specifications for *mkInst* and *mkNext*, which are named *mkInstSpec* and *mkNextSpec*. A pipeline defines *mkInstSpec* and *mkNextSpec* for each of its segments.

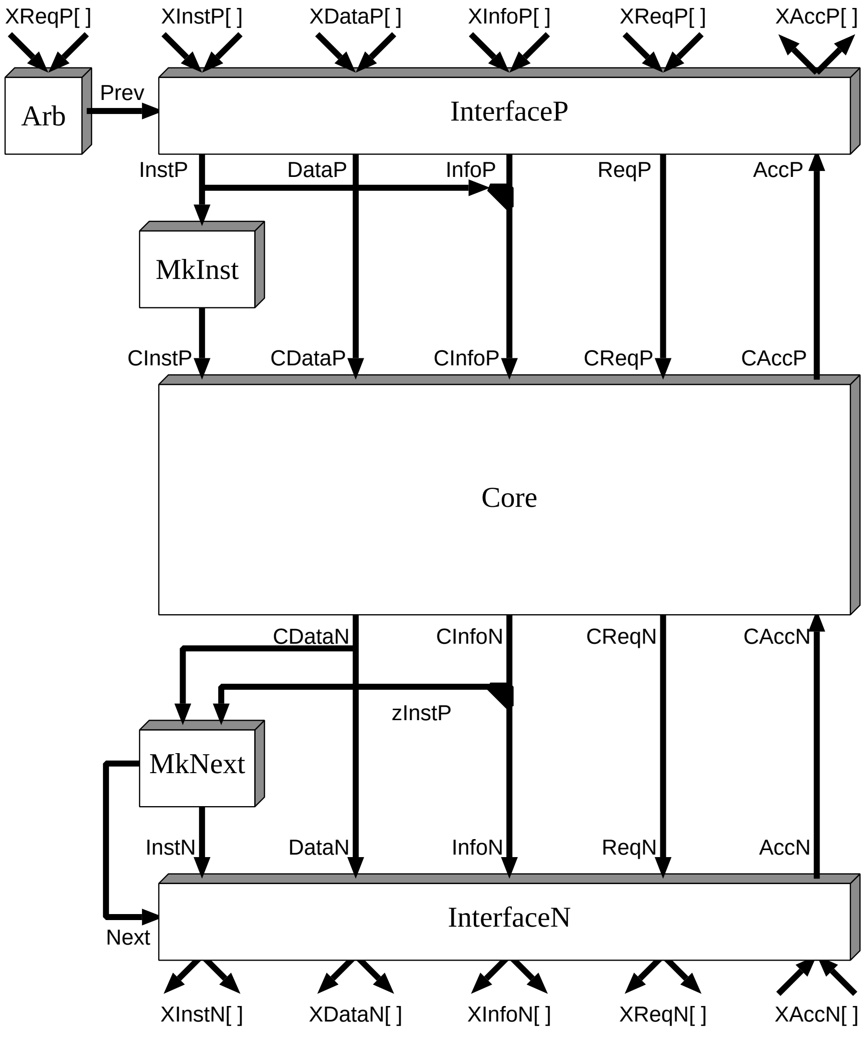


Figure 7.1: Segment with virtual functional units

The framework specifications for control schemes describe required properties for *mkInstSpec* and *mkNextSpec*. The definition of pipelining correctness is the same for all pipelines. This means that the verification of an individual pipeline does not define specifications for its arbitration and interface circuitry. The specifications for these units are included in the arbitration and control specifications for the framework parameters. Table 7.1 summarizes the virtual functional units and specifications that must be defined for each segment. Section 7.6 presents the formal specifications of the framework parameters.

Table 7.1: Functional Units for a Generic Segment

Type of circuits	Implementation	Specification
Pipelining	*Arb*	
	InterfaceP	
	InterfaceN	
Control	*MkInst*	*MkInstSpec*
	MkNext	*MkNextSpec*
Datapath	*Core*	*Comb*

Segments are connected by *bundles* of signals. Each segment has an array of bundles for input transactions and an array of bundles for output transactions. Each element in an array is connected to one other segment. The names of these arrays of external bundles are prefixed by "*X*" and suffixed by either "*P*" for previous (input) or "*N*" for next (output). For example, *XBundleP*[s_0] is the bundle from segment s_0 to the current segment. The bundles *CBundleP* and *CBundleN* are the bundles sent to the core of the segment and produced by the core. Table 7.2 lists the bundles for a segment. Along with the bundles, there are two other signals in a generic segment: *Prev* is the index of the selected previous segment and *Next* is the index of the selected next segment.

Table 7.2: Bundles in a segment

XBundleP[]	Array of bundles from previous segments
BundleP	Bundle from selected previous segment
CBundleP	Input bundle to core
CBundleN	Output bundle from core
BundleN	Bundle to next segment
XBundleN[]	Array of bundles to next segments

In the implementation of a pipeline, a transaction (*Trans*) contains three fields: data stationary control (*Inst*), data (*Data*) and information (*Info*). The information signal (*Info*) carries data-stationary control from higher-level pipelines in the structural hierarchy and will be explained shortly. Together with a transaction we group the request and accept signals (*Req* and *Acc*) together to form a *Bundle* (Table 7.3). As a notational shorthand we refer to the signals within a bundle by preserving the prefix and suffix and replacing "*Bundle*" with the name of the signal (*e.g.* *XDataP*[*s*] for the data field in *XBundleP*[*s*]).

Table 7.3: Signals in a bundle

Trans	*Inst*	Data-stationary control
Trans	*Data*	Data
Trans	*Info*	Information
	Req	Request
	Acc	Accept

The only exception to Table 7.3 is that the core does not produce any data-stationary control. Thus, there is no *CInstN* signal. This is because a core does not know what environments it will be used in. It does not know the segment to send a transaction to or the instruction to send with the transaction. The next segment and instruction are determined by *MkNext*.

As shown in Figure 7.1, we use the arbitration unit (*Arb*) to select which incoming request (*XReqP*[]) has the highest priority. The interface unit (*InterfaceP*) connects the bundle from the selected previous segment to the internal bundle *BundleP*. The interface units are usually implemented as multiplexers. The control unit *MkInst* uses the data-stationary control information from the incoming transaction (*InstP*) to determine the operation the core is to perform (*CInstP*).

The control unit *MkNext* determines the next segment to go to (*Next*) and the data-stationary control information to pass along with the data (*InstN*). The values of *Next* and *InstN* may be dependent upon the data-stationary control information that the transaction entered the segment with (*InstP*). In hierarchical pipelines, the core of a segment may itself be a pipeline. We use the signal *CInfoP* to send *InstP* with the transaction through the core of the segment, where it emerges as *CInfoN*. We split off the *InstP* field from *CInfoN*, name it *zInstP*, and connect it to *MkNext*. Because the current pipeline may be part of a larger pipeline, we pass the information fields from the higher-level pipelines through the current segment. The information fields from the higher levels use the *InfoP* and *InfoN* signals in the segment. We concatenate *InstP* for the current segment with the incoming *InfoP* to create *CInfoP*.

Combinational representations are discussed in detail in Section 7.4, here we briefly introduce them and relate them to Figure 7.1. The transactions in a combinational representation contain data-stationary control and data fields, but not an information field. Information fields are used in implementations, because multiple transactions may be in the core simultaneously and the core is responsible for keeping the data-stationary control and data together in a transaction. In the combinational representation we reason about each transaction individually. There is no difficulty in maintaining the connection between the data-stationary control and its data. Thus, there is no need for an information field in the transaction for a combinational representation. Also, because the combinational representation does not describe the pipelining aspects of behavior, there is no need for requests or accepts.

7.1.2 Temporal Reasoning: Definitions and Lemmas

A significant portion of the reasoning in the framework deals with the behavior of signals over time. In particular, we often need to prove that if one predicate is true, then a second will eventually become true. For example, one of the correctness criteria for a segment is that every request to enter the segment is eventually accepted. We state this formally as shown in Theorem 7.1: at any time (t_r) that a segment (s_0) requests another segment (s_1), there is a time (t_a) such that the request from s_0 to s_1 is held high from t_r to t_a and t_a is the first time after t_r that s_1 accepts a request from s_0. We use *holdUntil* (Definition 7.1) to capture the notion of holding a predicate ($\mathcal{P}_0$) true from some time (t_0) until $\mathcal{P}_1$ is true, where t_1 is the first time after t_0 that $\mathcal{P}_1$ is true (Figure 7.2). In Theorem 7.1 this allows us to uniquely match a request with a subsequent accept.

Theorem 7.1: *Every request is eventually accepted (Copy of Theorem 8.5)*
$\vdash \forall\ s_0,\ s_1,\ t_r\ .\ \exists\ t_a\ .\ \mathit{holdUntil}\ (\mathit{req}\ (s_0,\ s_1))\ t_r\ (\mathit{acc}\ (s_0,\ s_1))\ t_a$

Definition 7.1: *Hold a predicate until a second predicate is true*
$\mathit{holdUntil}\ \mathcal{P}_0\ t_0\ \mathcal{P}_1\ t_1\ \ \hat{=}$
$(\mathit{nextTimeTrue}\ \mathcal{P}_1\ t_0\ t_1)\ \&\ (\forall\ t : \{t_0..t_1\}\ .\ \mathcal{P}_0\ t)$

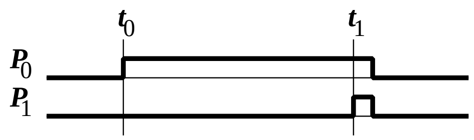


Figure 7.2: Depiction of *holdUntil*

We use *nextTimeTrue* (Definition 7.2) to say that a time (t_1) is the first time after t_0 that $\mathcal{P}_1$ is true. In *holdUntil*, if $\mathcal{P}_1$ is true at time t_0, then $t_0 = t_1$ and the left hand side of the conjunction in *nextTimeTrue* is vacuously true, because the range $\{t_0..t_1{-}1\}$ is empty.
To prove that *holdUntil* is true for two predicates ($\mathcal{P}_0$ and $\mathcal{P}_1$), we separate the obligations for $\mathcal{P}_0$ and $\mathcal{P}_1$. As shown in Lemma 7.1, *sustain* (Definition 7.3) describes the required

Definition 7.2: *nextTimeTrue*

$$\begin{aligned}&\mathit{nextTimeTrue}\ \mathcal{P}\ t_0\ t_1 \mathrel{\hat{=}}\\ &\quad (\forall\ t\ .\ \{t_0..t_1{-}1\}\ .\ \neg(\mathcal{P}\ t))\ \ \&\ \ \mathcal{P}\ t_1\end{aligned}$$

behavior of $\mathcal{P}_0$ and *holdCond* (Definition 7.4) describes the required behavior of $\mathcal{P}_1$. The definition of *sustain* says that once $\mathcal{P}_0$ becomes true, it will remain true until $\mathcal{P}_1$ becomes true. For $\mathcal{P}_1$, we use *holdCond* to say that if $\mathcal{P}_0$ is held true for a sufficient length of time, then $\mathcal{P}_1$ becomes true.

Lemma 7.1: *Hold until*

$$\begin{aligned}&\vdash\ \forall\ \mathcal{P}_0, \mathcal{P}_1\ .\\ &\quad (\mathit{sustain}\ \mathcal{P}_0\ \mathcal{P}_1)\ \&\\ &\quad (\forall\ t_0\ .\ \exists\ t_1\ .\ \mathit{holdCond}\ \mathcal{P}_0\ t_0\ P_1\ t_1)\ \Longrightarrow\\ &\qquad \forall\ t_0\ .\ \exists t_1\ .\ \mathit{holdUntil}\ \mathcal{P}_0\ t_0\ \mathcal{P}_1\ t_1\end{aligned}$$

Definition 7.3: *A sustained predicate*

$$\mathit{sustain}\ \mathcal{P}_0\ \mathcal{P}_1\ \mathrel{\hat{=}}\ \forall t\ .\ (\mathcal{P}_0\ t)\ \&\ \neg(\mathcal{P}_1\ t)\ \Longrightarrow\ \mathcal{P}_0\ (t{+}1)$$

We separate the obligations for $\mathcal{P}_0$ and $\mathcal{P}_1$, because the two predicates are usually signals that originate from different functional units. For example, when proving that every request is eventually accepted, Lemma 7.1 allows us to separate the obligations into showing that the request is held high until it is accepted and that if the request is held high for a sufficient length of time, then it will be accepted.

7.2 Correctness of Pipelines

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.

In the predicate *satisfies*, shown in Definition 7.5 and displayed using the symbol "$\sqsupset$", we separate the first two criteria, which describe pipelining aspects of correctness, from the third criteria, which describes the functional relationship between corresponding input and output transactions

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 7.4: *Hold a signal conditionally*

$$\begin{array}{l} \mathit{holdCond}\ \mathcal{P}_0\ t_0\ \mathcal{P}_1\ t_1\ \ \hat{=} \\ \quad (\forall\ t : \{t_0..t_1\}\ .\ \mathcal{P}_0\ t) \implies \mathcal{P}_1\ t_1 \end{array}$$

Definition 7.5: *Pipeline satisfies a specification*

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

definition uses *PipeliningOk* (Definition 7.8) to ensure that the pipelining aspects of correctness are met. The second part ensures that the functionality aspects of correctness are met. The correctness criteria for functionality says that when a matching relation (*Match*) matches an 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*). The function *getTransComb* (Definition 7.6) extracts the data-stationary control field (*Inst*) and the data field (*Data*) from a bundle containing data-stationary control, data, information, request, and accept fields.

Definition 7.6: *Extract transaction from a bundle of fields*

$$\mathit{getTransComb}\ ((\mathit{Inst},\ \mathit{Data},\ \mathit{Info}),\ \mathit{Req},\ \mathit{Acc})\ \ \hat{=}\ \ (\mathit{Inst},\ \mathit{Data})$$

The definition of *satisfies* is parameterized by a matching relation (*Match*), because the temporal relationship between input and output transactions varies from pipeline to pipeline. In some pipelines all transactions have the same latency (a uniform-latency ordering scheme, as described in Section 3.2.4), so all transactions exit a given number of clock cycles after they enter. In more complex pipelines, such as super-scalar instruction and floating-point pipelines, it is difficult to predict the latency for an individual transaction. In many of these pipelines transactions with the same destination register exit in-order and we use this fact to match input/output transactions (a tags-in-order ordering scheme where the transactions are tagged by their destination register).

A functional specification (*Spec*), is a relation over an input transaction and an output transaction and defines the computation that the pipeline is meant to perform. It is combinational and is not concerned with pipelining aspects of correctness. The pipelining correctness criteria are the same for every pipeline and are defined by *PipeliningOk*.

**For pipelines with more than one entry or exit port, t_i and t_o are pairs of type $\mathit{time} \times \mathit{port}$, where *port* is an entry or exit port for the pipeline. This is demonstrated with the superscalar instruction pipeline of the Adirondack in Section 6.4.

Some aspects of correctness are dependent on the environment of the pipeline working correctly. For example, if the environment does not agree to accept every transaction's request to exit the pipeline, then the pipeline can not guarantee that every transaction will exit the pipeline. The predicate *envOk* (Definition 7.7) allows us to say that a pipeline is only required to work correctly if its environment is correct. Some pipelines may place additional constraints on their environments, such as not issuing certain instructions in consecutive clock cycles. To capture this, *satisfies* is parameterized by *Constraints*, which can be used to restrict the behavior of environments.

Definition 7.7: *Correctness criteria for environments*

$$
\begin{aligned}
&\mathit{EnvOk}\ (\mathit{TransP},\ \mathit{ReqP},\ \mathit{AccP})\ (\mathit{TransN},\ \mathit{ReqN},\ \mathit{AccN})\ \hat{=}\\
&\quad \mathit{sustain}\ \mathit{ReqP}\ \mathit{AccP}\ \&\\
&\quad \forall t_r\ .\ \exists\ t_a\ .\ \mathit{holdCond}\ \mathit{ReqN}\ t_r\ \mathit{AccN}\ t_a
\end{aligned}
$$

As described in Lemma 7.1 in Section 7.1.2, we divide the responsibility of transferring a transaction between the two segments participating in the transfer. The first segment is responsible for sustaining its request and the second segment is responsible for accepting any request that is sustained. This division of responsibilities also applies between pipelines and environments. In the definition of *satisfies*, the pipeline (*Pipe*) interacts with an environment (*Env*) through an input stream (I) and an output stream (O). The predicate *envOk* (Definition 7.7) ensures that *Env* is a valid environment. In *envOk* we impose two requirements on the environment to ensure that every request to exit the environment is accepted. The first requirement uses *holdCond* to say that the environment accepts every sustained request. This is the same requirement that we place on segments within the pipeline as part of *CoreOk* (Definition 7.13). The second requirement is that the environment sustains its requests to transfer transactions to the pipeline; this also applies to segments within the pipeline.

In *satisfies*, the parameters to *envOk* are an input stream (I) and an output stream (O). Recall from Section 7.1.1 that the segments of the pipeline are connected to each other and to the environment through bundles of signals. These signals include a transaction, a request, and an accept. In the declaration of *envOk* we use pattern matching to separate the request and accept signals from the rest of the signals in the input and output bundles. The request and accept signals (*ReqP*, *AccP*, *ReqN*, and *AccN*) in *envOk* are examples of the virtual signals in our model of a pipeline. The virtual signals reflect actual pipelines and do not restrict implementations. In all pipelines that we have found, *ReqP* can either be defined to be always true (*i.e.* a new transaction is issued in every clock cycle) or be defined in terms of a signal in the implementation (*e.g.* instruction cache hit). The counterpart to *ReqP* is *AccP*, which represents when a pipeline is able to accept an incoming transaction. As an example, we sometimes define *AccP* to be the absence of a stall.

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
- every transaction that exits the pipeline has entered

These criteria are captured in *PipeliningOk* (Definition 7.8). The relation *canEnter* (Definition 7.9) 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*, Definition 7.10). This ensures that every input transaction eventually exits and every output transaction entered the pipeline. These properties guarantee that the pipeline is free from deadlock and livelock and that transactions are not created inside the pipeline.

Definition 7.8: *Pipelining correctness criteria*

$$\begin{array}{l} \mathit{PipeliningOk}\ \mathit{Match}\ \ I\ \ O\ \hat{=} \\ \quad (\mathit{canEnter}\ I\ O)\ \&\ (\mathit{isOneToOne}\ \mathit{Match}\ I\ O) \end{array}$$

Definition 7.9: *Correctness criteria to ensure that transactions are able to enter pipeline*

$$\begin{array}{l} \mathit{canEnter}\ (\mathit{TransP},\ \mathit{ReqP},\ \mathit{AccP})\ (\mathit{TransN},\ \mathit{ReqN},\ \mathit{AccN})\ \ \hat{=} \\ \quad \mathit{sustain}\ \mathit{ReqN}\ \mathit{AccN}\ \& \\ \quad \forall t_r\ .\ \exists t_a\ .\ \mathit{holdCond}\ \mathit{ReqP}\ t_r\ \mathit{AccP}\ t_a \end{array}$$

The definition of *envOk* describes the requirements that we impose on environments: every sustained request is accepted and every request is sustained. We enforce these same two requirements on pipelines in *canEnter* (Definition 7.9). These two definitions are mirrors of each other and together they ensure that every request from the environment to enter the pipeline is accepted and every request from the pipeline to enter the environment is accepted.

Definition 7.10: *One to one mapping between input and output transactions*

$$\begin{array}{l} \mathit{isOneToOne}\ \mathit{Match}\ I\ O\ \ \hat{=} \\ \quad (\forall t_i\ .\ \mathit{XfrIn}\ I\ t_i\ \ \Longrightarrow\ \exists! t_o\ .\ \mathit{Match}\ I\ O\ t_i\ t_o)\ \& \\ \quad (\forall t_o\ .\ \mathit{XfrOut}\ O\ t_o\ \ \Longrightarrow\ \exists! t_i\ .\ \mathit{Match}\ I\ O\ t_i\ t_o)\ \& \\ \quad (\forall t_i,\ t_o\ .\ \mathit{Match}\ I\ O\ t_i\ t_o\ \ \Longrightarrow\ \ (\mathit{XfrIn}\ I\ t_i)\ \&\ (\mathit{XfrOut}\ O\ t_o)) \end{array}$$

The first two clauses in *isOneToOne* (Definition 7.10) ensure that every input transaction matches exactly one output transaction and that every output transaction matches exactly one input transaction. The third clause says that the matching relation accurately reflects when transactions enter and exit the pipeline. The predicates *XfrIn* and *XfrOut* (Definitions 7.27 and 7.28 in Section 7.6.2.4) represent when a transaction enters and leaves a pipeline.

7.3 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 7.1).

$$(\mathit{Pipe},\ \mathit{Match},\ \mathit{Constraints}) \sqsupset \mathit{Spec} \tag{7.1}$$

As an abbreviation, we package the implementation of our example pipeline (*Pipe*), matching relation (*Match*), and constraints (*Constraints*) as *PipeRecord* (Definition 7.11).

Definition 7.11: *Pipe record*
$\mathit{PipeRecord} = (\mathit{Pipe},\ \mathit{Match},\ \mathit{Constraints})$

In a direct proof of Equation 7.1, we 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 7.3). 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.

$\vdash\ (\mathit{Pipe},\ \mathit{Match},\ \mathit{Constraints}) \sqsupset \mathit{Spec}$

BY `Unfold` *satisfies* `(*` Definition 7.5 `*)`

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

- $(\mathit{Pipe}\ I\ O)\ \&\ (\mathit{Constraints}\ I\ O)\ \&\ (\mathit{envOk}\ I\ O)$
  $\vdash\ \mathit{Match}\ I\ O\ t_i\ t_o \implies$
  $\quad \mathit{Spec}\ (\mathit{getTransComb}\ (I\ t_i))\ (\mathit{getTransComb}\ (O\ t_o))$

Figure 7.3: Direct verification of canonical example; unfold *satisfies*

In Section 7.3.1 we introduce the *combinational representation* of a pipeline and show how we use this to completely separate the pipelining and functionality concerns when verifying a pipeline. This leaves us with a functionality proof that is purely combinational and a pipelining proof that is largely data-independent. In Section 7.3.2 we simplify the pipelining proof using the specifications for the four parameters to the framework (protocol, arbitration, control, and ordering).

7.3.1 Using A Combinational Representation To Verify A Pipeline

To remove all aspects of pipelining from the functionality proof we use Theorem 7.2 to introduce a *combinational-logic representation* (*Comb*) of the pipeline. A combinational representation is a specially constructed specification residing between the concrete implementation (*Pipe*) and the abstract specification (*Spec*). 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. Section 7.4 describes calculating combinational representations and demonstrates calculating the combinational representation of the *diffAdd* pipeline, which was described in Chapter 5. The proof of Theorem 7.2 is done by unfolding the definition of *satisfies* and applying Modus Ponens.

Theorem 7.2: *Transitivity of satisfies*

$$
\begin{array}{l}
\vdash \forall \mathit{PipeRecord},\ \mathit{Comb},\ \mathit{Spec}\ . \\
\qquad \mathit{PipeRecord} \sqsupset \mathit{Comb}\ \& \\
\qquad (\forall T_i,\ T_o\ .\ \mathit{Comb}\ T_i\ T_o \implies \mathit{Spec}\ T_i\ T_o) \implies \\
\qquad\quad \mathit{PipeRecord} \sqsupset \mathit{Spec}
\end{array}
$$

Using Theorem 7.2 to verify a pipeline, we introduce the combinational representation and have the two goals shown in Figure 7.4, rather than those shown in Figure 7.3. The first goal, which we refer to as the pipelining goal, is to 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 7.3. 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 7.4, which we refer to as the functionality goal, is to show that the combinational representation implies the specification.[***] This goal is purely combinational and can be solved using standard hardware-verification techniques.

7.3.2 Using the Framework to Verify a Pipeline

In the previous section (Section 7.3.1), we introduced the combinational representation of a pipeline and used Theorem 7.2 to separate the proof that a pipeline *satisfies* a specification

[***] 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.

$\vdash$ *PipeRecord* $\sqsupset$ *Spec*

BY `Theorem 7.2`

$\vdash$ *PipeRecord* $\sqsupset$ *Comb* (* pipelining *)

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

Figure 7.4: Verification of canonical example; introduce combinational representation

into a proof that the pipeline *satisfies* the combinational representation (the pipelining proof) and a proof that the combinational representation implies the specification (the functionality proof). In this section we simplify the pipelining proof.

We use Theorem 7.3 to divide the proof goal that a pipeline *satisfies* its combinational representation (the first subgoal in Figure 7.4) into four subgoals (Figure 7.5): the core (datapath) for each segment is valid (*CoresOk*, Definition 7.12), the constraints of the segments are met (*ConstraintsOk*, Definition 7.14), the combinational representation is an accurate representation of the pipeline (*CombOk*, Definition 7.15), and the pipelining circuitry that glues the segments together is valid (*FworkParamsOk*, Definition 7.16). Theorem 7.3 is the centerpiece of the framework; its proof is described in Chapter 8. In the rest of this section we describe the relations used in Theorem 7.3. Section 7.3.3 summarizes the proof obligations incurred when verifying a pipeline.

Theorem 7.3: *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*)) $\implies$
(*Pipe, Match, Constraints*) $\sqsupset$ *Comb*

As described in Section 7.1.1, we reason about generic segments, each of which contains a core and a number of virtual functional units and signals. We work with hierarchical pipelines, where the core of a segment may itself be a pipeline. Thus, each segment (s) has a core (*core s*), matching relation (*match s*), constraints (*constraints s*), and combinational representation (*comb s*). We package the core, matching relation, and constraints as a tuple (*coreRecord*). The definition of *CoresOk* (Definition 7.12) uses *CoreOk* (Definition 7.13, displayed as "$\boxtimes$") to say that each core in the pipeline is valid. We use the SML

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

BY `Theorem 7.3`

$\vdash$ *CoresOk Pipe*

$\vdash$ *ConstraintsOk Pipe Constraints*

$\vdash$ *CombOk Pipe Comb*

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

Figure 7.5: Verification of canonical example; use framework to show that pipeline *satisfies* combinational representation

construct *open* to add the segments and their associated core records and combinational representations to the scope of the body of *local-in-end*.[†]

Definition 7.12: *Cores of segments are correct*

CoresOk Pipe $\hat{=}$
local open Pipe in
$\forall s \,.\, (coreRecord\ s) \boxed{\triangleright} (comb\ s)$
end

The definition of *CoreOk* (Definition 7.13) says that the core (*Core*), matching relation (*Match*), and constraints (*Constraints*) *satisfy* their combinational representation (*Comb*) and the core meets four requirements that enable us to add pipelining and control circuitry to create a valid segment. These requirements are that the combinational representation is functional (it produces equal outputs for equal inputs), the data in an output transaction is held constant while it is waiting for its request to be accepted, the information fields in a pair of matching input and output transactions are equal, and the latency through the core is non-negative.
In *CoreOk* we capitalize the variables and do not parameterize them by a segment identifier (*e.g. Core* rather than *core s*), because *CoresOk* selects which segment is being described by instantiating *coreRecord* and *comb* with the given segment (s) before calling *CoreOk*.

[†]For notational convenience in Definitions 7.12, 7.14, and 7.15 we do not make a distinction between the variable *Pipe* as a relation and as a structure defining a set of segments.

Definition 7.13: *(CoreOk) Core of segment is correct*

$$
\begin{array}{l}
(\mathit{Core},\ \mathit{Match},\ \mathit{Constraints}) \boxtimes \mathit{Comb}\ \hat{=} \\
\quad (\mathit{Core},\ \mathit{Match},\ \mathit{Constraints}) \sqsupset \mathit{Comb}\ \& \\
\quad (\forall T_i,\ T_o,\ T_o' \ .\ (\mathit{Comb}\ T_i\ T_o)\ \&\ (\mathit{Comb}\ T_i\ T_o') \implies (T_o = T_o'))\ \& \\
\quad \forall I,\ O\ .\ (\mathit{Core}\ I\ O)\ \&\ (\mathit{Constraints}\ I\ O)\ \&\ (\mathit{envOk}\ I\ O) \implies \\
\qquad \mathit{let}\ \ (\mathit{TransN},\ \mathit{ReqN},\ \mathit{AccN}) \ \hat{=}\ O\ \ \mathit{in} \\
\qquad\quad \forall\ t\ .\ (\mathit{ReqN}\ t)\ \&\ \neg(\mathit{AccN}\ t) \implies \mathit{TransN}\ (t{+}1) = \mathit{TransN}\ t\ \& \\
\qquad\quad \forall t_i,\ t_o\ .\ \mathit{Match}\ I\ O\ t_i\ t_o \implies \\
\qquad\qquad (\mathit{getInfo}\ (O\ t_o)) = (\mathit{getInfo}\ (I\ t_i))\ \& \\
\qquad\qquad t_i \le t_o \\
\qquad \mathit{end}
\end{array}
$$

The requirement that the combinational representation is functional is combined with the requirement in *satisfies* (Definition 7.5) that the core implies the combinational representation and with the control specifications, which state similar requirements for the implementations and specifications of the control circuitry, to prove that transactions follow the same path through the implementation and combinational representation of the pipeline. The third clause in *CoreOk* says that the output transaction (*TransN*) does not change while a request is waiting to be accepted. A transaction's choice of a next segment may be dependent on data-stationary control that the transaction entered the segment with (which is part of the information field of the transaction) and/or the data in the transaction (data-dependent control). Allowing the contents of the transaction to change could cause the choice of the next segment to change, and so the request from the current segment to the next segment would not be sustained.

We require that the information field is not modified by the core, because the information field is used to keep data-stationary control with data as it traverses through the core. This requirement is included as part of *CoreOk* and not *satisfies*, because we use the information field in a pipeline only if the pipeline is to be used as the core of a segment in a higher level pipeline. The relation *satisfies* is used for all levels of pipelines and combinational representations. Top level pipelines do not have information fields. This requirement is not included as part of either the combinational representation or the user supplied functional specification, because combinational representations and specifications do not have access to the information field. The information field is part of the pipelining aspects of a design; the combinational representation and user supplied specification reflect only the datapath and control circuitry.

The last requirement in *CoreOk* is that the latency through the core is always non-negative. Restricting ourselves to non-negative latencies means that time monotonically increases as a transaction traverses through a pipeline. This greatly simplifies the proof that every output transaction results from exactly one input. For implementable pipelines, non-negative latencies are obviously a reasonable requirement. Some abstract models of pipelines use *anti-delays*, which are negative delays. Anti-delays appear most frequently in retiming, which is used for optimization and synthesis [She86, LB90]. Anti-delays are added to pipelines such that each anti-delay can be composed with a delay element or pushed to the exterior of the pipeline, thereby removing the anti-delays from the pipeline when it is built. This technique is compatible with the requirement that the core has a non-negative latency.

The definition of *satisfies* is parameterized by a set of constraints. A pipeline is only required to work correctly if its environment meets these constraints. This means that when working with hierarchical pipelines, a segment may have constraints that its environment must meet. Thus, Theorem 7.3 includes the requirement that the constraints of the segments are met (*ConstraintsOk*, Definition 7.14). The environment of a segment is the other segments in the pipeline and the environment of the pipeline. In *ConstraintsOk*, *cBundleP s* and *cBundleN s* are the bundles of input and output signals to the core of the segment; they are analogous to the input and output streams I and O for a pipeline. Section 7.1.1 described the bundles for a segment. The definition of *ConstraintsOk* uses the SML construct *local-open-in-end*, just as *CoresOk* does, to add the segments and the bundles connecting the segments to the current scope.

Definition 7.14: *Constraints of segments are met*

$$
\begin{array}{l}
\mathit{ConstraintsOk}\ \mathit{Pipe}\ \mathit{Constraints} \mathrel{\hat{=}} \\
\quad \forall\ I,\ O\ .\ (\mathit{Pipe}\ I\ O)\ \&\ (\mathit{Constraints}\ I\ O)\ \&\ (\mathit{envOk}\ I\ O) \implies \\
\qquad \mathit{local\ open\ Pipe\ in} \\
\qquad\quad \forall\ s\ .\ \mathit{constraints}\ s\ (\mathit{cBundleP}\ s)\ (\mathit{cBundleN}\ s) \\
\qquad \mathit{end}
\end{array}
$$

Definition 7.15: *Combinational representation is correct*

$$
\begin{array}{l}
\mathit{CombOk}\ \mathit{Pipe}\ \mathit{Comb} \mathrel{\hat{=}} \\
\quad \mathit{local\ open\ Pipe\ in} \\
\qquad \forall\ T_i,\ T_o\ .\ \mathit{Comb}\ T_i\ T_o \iff \mathit{travComb}\ (\mathit{Entry},\ T_i)\ (\mathit{Exit},\ (T_o,\ \mathit{Bot})) \\
\quad \mathit{end}
\end{array}
$$

The relation *CombOk* (Definition 7.15) relates the combinational representation to the pipeline. It requires that the combinational representation is equivalent to traversing the combinational representations of the segments. Section 7.4 describes how we use *travComb* (Definition 7.18) for traversing through the combinational representations of the segments and demonstrates the use of *travComb* to calculate the combinational representation of the *diffAdd* pipeline from Chapter 5.

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* (Definition 7.16) says that if the cores of the segments are valid, the constraints of the segments are met, and the pipeline's constraints on the environment are met then the pipeline meets the specifications of the framework parameters. Section 7.6 describes these specifications in detail.

Definition 7.16: *Specifications for framework parameters are satisfied*

$$
\begin{array}{l}
\quad FworkParamsOk\ (Pipe,\ Match,\ Constraints)\ \hat{=} \\
\quad\quad CoresOk\ Pipe\ \& \\
\quad\quad ConstraintsOk\ Pipe\ Constraints\ \Longrightarrow \\
\quad\quad\quad \forall\ I,\ O\ .\ (Pipe\ I\ O)\ \&\ (Constraints\ I\ O)\ \&\ (envOk\ I\ O)\ \Longrightarrow \\
\quad\quad\quad\quad \begin{array}{lll} ProtocolOk & Pipe & \& \\ ArbitrationOk & Pipe & \& \\ ControlOk & Pipe & \& \\ OrderingOk & Pipe\ Match & \end{array}
\end{array}
$$

7.3.3 Summary of Verifying a Pipeline

When verifying that a pipeline *satisfies* a specification we use Theorem 7.2 to introduce a combinational representation and separate the pipelining and functionality aspects of verification. We then use Theorem 7.3 to simplify the pipelining proof. This leaves us with the five proof obligations listed in Table 7.4.

Table 7.4: Proof obligations when verifying a pipeline

CoresOk Pipe
  Datapaths and control circuitry of segments are valid

ConstraintsOk Pipe Constraints
  Constraints of segments are met

CombOk Pipe Comb
  Combinational representation accurately represents the pipeline

FworkParamsOk (*Pipe, Match, Constraints*)
  Pipeline meets specifications of framework parameters

$\forall\ T_i,\ T_o\ .\ Comb\ T_i\ T_o\ \Longrightarrow\ Spec\ T_i\ T_o$
  Combinational representation implies specification

The first obligation uses *CoresOk* (Definition 7.12) to say that the core of each segment is valid. The definition of *CoresOk* says that each core *satisfies* its combinational representation and meets several other conditions. To prove that a core meets the additional requirements we use Theorem 7.4, which has the same antecedents as Theorem 7.3, but has the stronger consequent that the pipeline meets all of the requirements in *CoreOk*. This means that the pipeline can be used as the core of a segment in higher levels of the structural hierarchy. By using the framework to verify pipelines, the first goal in Table 7.4 is solved in the normal progression through the structural hierarchy. The proof of Theorem 7.4 appears in Chapter 8.

Theorem 7.4: *Sufficiency of conditions for a valid core*

$$
\begin{array}{l}
\vdash \forall\ \mathit{Pipe},\ \mathit{Match},\ \mathit{Constraints},\ \mathit{Comb}\ . \\
\qquad (\mathit{CoresOk}\ \mathit{Pipe})\ \& \\
\qquad (\mathit{ConstraintsOk}\ \mathit{Pipe}\ \mathit{Constraints})\ \& \\
\qquad (\mathit{CombOk}\ \mathit{Pipe}\ \mathit{Comb})\ \& \\
\qquad (\mathit{FworkParamsOk}\ (\mathit{Pipe},\ \mathit{Match},\ \mathit{Constraints}))\ \Longrightarrow \\
\qquad\quad (\mathit{Pipe},\ \mathit{Match},\ \mathit{Constraints})\ \boxed{\triangleright}\ \mathit{Comb}
\end{array}
$$

The second goal in Table 7.4 is to prove that the constraints for each segment are met. This obligation is usually quite easy to solve. Many segments do not put any constraints on their environment. In all pipelines that we have found that have segments that constrain their environment, each constraint affects only one segment (or the environment) and the constraints are direct consequences of the behavior of the segment (or environment). For example, the integer and floating-point pipelines of most microprocessors use a tags-unique ordering scheme where transactions are tagged with their destination register. This requires that the environment issues transactions so that at most one transaction with a given destination is active at any time. To ensure that transactions use the correct data values, the register files of the microprocessors "lock" a register while it is waiting for a result to be computed. This guarantees that no subsequent transaction that will write to the register will be issued to the integer or floating-point pipeline until the previous transaction using that register has finished. This prevents more than one transaction with the same destination register from being in the pipeline at the same time, and so the constraint is satisfied.

The third goal in Table 7.4 is to prove that the combinational representation correctly represents the pipeline. The requirement is that the combinational representation is equivalent to traversing the combinational representations of the segments. As shown in Section 7.4, this is exactly how we calculate combinational representations, so this goal is trivially solved. The fourth goal is to prove that the specifications of the framework parameters are met. Chapter 4 describes how to use a set of decision trees to informally verify that a pipeline meets the specifications. The formal specifications of the parameters are presented in Section 7.6.

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.

7.4 Calculating a Combinational Representation

In this section we describe how to calculate the combinational representation of a pipeline. Combinational representations are used as an intermediate level representation between the implementation and specification. They reflect a pipeline's datapath and control circuitry, but not its pipelining circuitry. Combinational representations reflect the structure of pipelines, in that they are built by composing the combinational representations of segments. In the remainder of this section, we introduce the definitions needed to calculate a combinational representation and in Appendix B.1 we calculate the combinational representation of the *diffAdd* pipeline from Chapter 5.

The combinational representation of a pipeline (*Comb*) is defined as traversing a path through the combinational representations of the segments from the entry segment to the exit segment. (Definition 7.17).

Definition 7.17: *Combinational representation of a pipeline*

$$Comb\ T_i\ T_o \ \hat{=}\ travComb\ (Entry,\ T_i)\ (Exit,\ (T_o,\ Bot))$$

The relation for traversing a path between the combinational representations of segments (*travComb*) is defined inductively as shown in Definition 7.18. The first point in the path is identified by a pair of a segment (s_0) and an input transaction (T_0). The last point is identified by the last segment in the path (s_n), the output transaction from that segment (T_n), and the next segment to send the transaction to (s'). If the transaction is to exit the pipeline, the next segment is the bottom environment (*Bot*).

Definition 7.18: *Traverse through a combinational representation*

$$\begin{array}{l}
travComb\ (s_0,\ T_0)\ (s_n,\ (T_n,\ s'))\ \hat{=} \\
\quad (s_0 = s_n)\ \&\ (segComb\ s_0\ T_0\ (T_n,\ s')) \\
\quad \vee \\
\quad \exists s_x,\ T_x\ . \\
\qquad segComb\ s_0\ T_0\ (T_x,\ s_x)\ \& \\
\qquad travComb\ (s_x,\ T_x)\ (s_n,\ (T_n,\ s'))
\end{array}$$

In the base case of *travComb*, the first segment (s_0) and last segment (s_n) are the same ($s_0 = s_n$) and the combinational representation of the segment (*segComb* s_0) produces the output transaction T_n and selects s' as the next segment. In the inductive case, there is an intermediate segment (s_x) and transaction (T_x) such that the combinational representation of s_0 produces T_x and selects s_x as the next segment and traversing from s_x to s_n produces T_n and selects s'.

Each segment in a pipeline has a combinational representation, written as *segComb* s for segment s. As shown in Definition 7.19, the combinational representation of a segment is built from the specifications of the control units (*mkInstSpec* s and *mkNextSpec* s) and the combinational representation of the core of the segment (*comb s*). We use the specification of the control units rather than the implementations, because the specifications are relations over values, while the specifications are relations over signals. This makes *segComb* independent of time.

Definition 7.19: *Combinational representation of a segment*
$$
\begin{array}{l}
\mathit{segComb}\ s\ (\mathit{InstP},\ \mathit{DataP})\ ((\mathit{InstN},\ \mathit{DataN}),\ s')\ \ \hat{=}\\
\quad \exists\ \mathit{CInstP}\ .\\
\qquad \mathit{mkInstSpec}\ s\ \mathit{InstP}\ \mathit{CInstP}\ \&\\
\qquad \mathit{comb}\ s\ (\mathit{CInstP},\ \mathit{DataP})\ \mathit{DataN}\ \&\\
\qquad \mathit{mkNextSpec}\ s\ (\mathit{InstP},\ \mathit{DataN})\ (\mathit{InstN},\ s')
\end{array}
$$

The input to *segComb* is a transaction consisting of a data-stationary control field (*InstP*) and data field (*DataP*). The output is the transaction with the data-stationary control field *InstN* and data *DataN* along with the identify of the next segment to transfer to (s'). The specification of the *mkInst* control unit uses *InstP* to generate the operation that the core is to perform (*CInstP*). The combinational representation of the core (*comb s*) performs *CInstP* on *DataP* and produces *DataN*. The data-stationary control that the transaction entered with (*InstP*) and the data produced by the core (*DataN*) are passed to *mkNextSpec*, which produces *InstN* and s'.

7.5 Traversing a Path Through the Implementation of a Pipeline

Section 7.4 described how we use *travComb* to calculate a combinational representation by traversing the combinational representations of the segments. In this section we describe traversing a path through an implementation using the relation *travPipe* (Definition 7.20). The relation *travPipe* is one of the critical definitions used in reasoning about the framework. When verifying a pipeline we encounter *travPipe* in the ordering specification, which says that the matching relation for a pipeline matches two times (t_i and t_o) if and only if the transaction that enters the pipeline at t_i traverses and exits at t_o. We demonstrate the use of *travPipe* with the *diffAdd* pipeline from Chapter 5 in Appendix B.2 and with a pipeline that contains a path with a loop in Appendix C.

7.5.1 Traversing Implementations in General

The relation *travPipe* (Definition 7.20) is defined inductively. A transaction traverses from segment s_0 at time t_0 to s_n at t_n ($\mathit{travPipe}\ (s_0,\ t_0)\ (s_n,\ t_n)$) if s_0 and s_n are the same segment and the transaction uses s_0 from t_0 to t_n ($\mathit{match}\ s_0\ (t_0,\ t_n)$); or if there is an intermediate segment (s_x) and time (t_x) such that the transaction uses s_0 from t_0 to t_x, transfers from s_0 to s_x at t_x, and traverses from s_x at t_x to s_n at t_n.

Definition 7.20: *Traverse through the implementation of a pipeline*

$$
\begin{array}{l}
\mathit{travPipe}\ (s_0,\ t_0)\ (s_n,\ t_n)\ \hat{=} \\
\quad (s_0 = s_n)\ \&\ (\mathit{match}\ s_0\ (t_0,\ t_n)) \\
\quad \vee \\
\quad \exists\ s_x,\ t_x\ . \\
\qquad \mathit{match}\ s_0\ (t_0,\ t_x)\ \& \\
\qquad \mathit{xfr}\ (s_0,\ s_x)\ t_x\ \& \\
\qquad \mathit{travPipe}\ (s_x,\ t_x)\ (s_n,\ t_n)
\end{array}
$$

A transaction uses a segment (s_0) from time t_0 to t_n means that the transaction enters s_0 at t_0, exits at t_n, and is inside the segment the entire time between t_0 and t_n. The matching relation for a segment detects when a transaction enters and exits the segment. If the framework is used at all levels of the structural hierarchy, then when a matching relation for a segment matches two times (t_0 and t_n) we can guarantee that the transaction is inside the segment the entire time between t_0 and t_n.

The core of each segment is either a single stage or is a pipeline that was verified using the specifications of the framework parameters. In both cases, the matching relation for the segment matches two times (t_0 and t_n) if and only if the transaction traverses the pipeline from t_0 to t_n. For stages there is only one transaction in the core at a time. For segments with cores that are pipelines, the specification for ordering schemes (Section 7.6.2.4) guarantees this property.

Appendix B.2 illustrates traversing through the implementation of the *diffAdd* pipeline from Chapter 5 and Appendix C does the same for a pipeline that contains paths with loops.

7.6 Specifications of Framework Parameters

7.6.1 Overview of Specifications

When deriving specifications for the framework parameters our goal was to capture as much reasoning as possible within the framework. This prevents the verification of an individual pipeline from containing reasoning that is duplicated in the verification of other pipelines. Another objective was for the framework to be amenable to mechanization in a proof-development system. For this reason, we want to be able to easily check pipelines against the specifications of the parameters by mechanical means. Mechanized systems are much better at managing large amounts of information than humans are. In contrast, humans are better at reasoning across large abstraction gaps. Because the framework is targeted at mechanization, we have a rather large number of specifications, but almost all of the specifications are relatively simple and easy to check.

We chose the four framework parameters (protocol, arbitration, control, and ordering) by examining how transactions flow through pipelines. We found that three of the parameters (protocol, arbitration, and control) are closely related to the pipelining circuitry in segments. To take advantage of this observation, we wrote the protocol, arbitration, control specifications in terms of a set of virtual functional units for each segment that implement the specified behavior. The functional units are described in Section 7.1.1. Verifying that a pipeline satisfies the specifications for the four parameters is done by defining the correspondence between the components of the real pipeline and the virtual functional units and then showing that the components in the real pipeline satisfy the specifications for the corresponding virtual functional units.

The protocol specifications say that the request and accept signals between segments are connected properly and that if two segments exchange a request and an accept, then the first segment must send its output transaction to the second. The arbitration specifications ensure that transactions are able to progress through the pipeline (prevention of deadlock and livelock). The control specifications guarantee that every path through the pipeline leads to an exit. This prevents a "correct" pipeline from containing paths that do not terminate. Finally, the ordering specification says that the matching relation for the pipeline matches corresponding input/output transactions.

The protocol specifications make up a significant portion of the total number of specifications (nine out of eighteen). The only protocol specification that requires any effort to check is that transactions are transferred between segments correctly, all of the others are very easy to check. There are three arbitration specifications, one of which is very easy to check; five control specifications, two of which are trivial to check and one of which is usually easy to check; and one ordering specification. Thus, out of eighteen specifications, there are only seven specifications that require much effort to check. These specifications are summarized in Table 7.5.

To help explain the specifications, within the discussion of the specifications we mention a variety of lemmas and theorems. As shown in Theorem 7.4, the verification of a pipeline must prove that the pipeline meets the specifications; the verification of a pipeline does not include proofs of the lemmas and theorems. We have proved the lemmas and theorems using the specifications and the definitions of *CoresOk*, *ConstraintsOk*, and *CombOk*. These proofs are part of the verification of the framework, which is presented in Chapter 8.

As described in Section 3.3, we have found a relatively small number of protocol, arbitration, control, and ordering schemes that are sufficient to characterize all of the pipelines that we have examined. These instantiations guide the proof that a pipeline meets the specifications. In addition, we are developing formal proofs of correctness for these schemes. Using these proofs will allow a pipeline to be verified against a set of specifications that are easier to check than the general specifications presented here. Each scheme will be verified by proving that the specifications it imposes on a pipeline are at least as strong as the general specifications. A pipeline can currently be informally verified using the decision trees for the framework parameters (Section 4.2), which guide the instantiation of each parameter and list the obligations that must be verified to ensure that the instantiation is valid.

Table 7.5: Summary of major specifications

Protocol	
Spec 7.3	Transactions are transferred between segments correctly.
Arbitration	
Spec 7.10	The highest priority request to each segment is eventually accepted
Spec 7.12	Every request to a segment becomes the highest priority request
Control	
Spec 7.13	All paths through the pipeline lead to an exit
Spec 7.14, 7.15	The implementations of the control circuitry imply their specifications
Ordering	
Spec 7.18	The matching relation for the pipeline finds corresponding input/output transactions

7.6.2 Formal Specifications of Framework Parameters

7.6.2.1 Protocol Specifications

The specifications for a protocol scheme are concerned with the connection of signals between segments and the transferring of transactions between segments. The specifications describe the behavior of signals connected to the interface units of a segment. In keeping with our philosophy of writing simple specifications that are easy to check mechanically, we divide the protocol specifications into three groups: the connection of signals between segments is correct, the interface unit at the top of the segment (*InterfaceP*) is correct, and the interface unit at the bottom of the segment (*InterfaceN*) is correct.

The specifications allow the interface units for each segment to be checked in isolation, rather than as part of the entire pipeline. This simplifies verification and makes it easier to isolate errors. We are able to verify the interface units in isolation, because we specify their behavior in terms of the ports on the units, rather than in terms of signals in the pipeline. The *InterfaceP* unit for segment s_1 has an array of ports that are connected to an array of signals, where each signal in the array is connected to one of the segments that can send transactions. We use square brackets ("[" and "]") to denote indexing into an array: $xReqP\ s_1$ is the array ports on the *InterfaceP* unit of s_1 for request signals to s_1 and $(xReqP\ s_1)[s_0]$ is the port on s_1 for the request from s_0 to s_1. There is a corresponding port on the *InterfaceN* unit for s_0, which is named $(xReqN\ s_0)[s_1]$. Specifications 7.1 and 7.2 say that corresponding request and accept ports are connected. Specification 7.3 says that the corresponding ports for transactions are connected. For requests and accepts we require that the corresponding ports are always connected while for transactions we only require that the ports are connected when s_0 requests s_1 and s_1 accepts the request from

s_0 (*i.e.* when we transfer a transaction).

Specification 7.1: *Protocol (Connection for request signals between segments)*

$$\forall\ s_0,\ s_1,\ t\ .\ (xReqP\ s_1)[s_0](t) \iff (xReqN\ s_0)[s_1](t)$$

Specification 7.2: *Protocol (Connection for accept signals between segments)*

$$\forall\ s_0,\ s_1,\ t\ .\ (xAccN\ s_0)[s_1](t) \iff (xAccP\ s_1)[s_0](t)$$

Specification 7.3: *Protocol (Connection for transactions between segments)*

$$(xReqN\ s_0)[s_1](t)\ \&\ (xAccP\ s_1)[s_0](t) \implies (xTransP\ s_1)[s_0](t) = (xTransN\ s_0)[s_1](t)$$

Specifications 7.4–7.6 describe the required behavior for the *InterfaceP* unit of a segment (s_1). Specifications 7.4 and 7.5 say that the *InterfaceP* unit sends a request to the core of the segment ($reqP\ s_1\ t$) when there is any incoming external request ($(xReqP\ s_1)[s_0](t)$) and that *InterfaceP* sends an accept to segment s_0 when the core is able to accept a transaction ($accP\ s_1\ t$) and s_0 is the selected previous segment ($s_0 = prev\ s_1\ t$).
Just as with transferring transactions between segments, in Specification 7.6 we require that the interface unit connects the incoming transaction ($(xTransP\ s_1)[s_0]$) to the input transaction to the core ($transP\ s_1$) when the core of s_1 can accept an incoming transaction ($accP\ s_1$) and s_0 is the selected previous segment ($prev\ s_1\ t = s_0$).
Specifications 7.7–7.9 describe the required behavior of the *InterfaceN* unit for segment s_0. Specification 7.7 says that the request signal to s_1 ($(xReqN\ s_0)[s_1]$) is true when there is a request from the core of s_0 ($xReqN\ s_0$) and s_1 is the selected next segment ($s_1 = next\ s_0\ t$). In Specification 7.8, the interface unit forwards an accept to the core ($accP\ s_0$) if the selected next segment sends an accept to s_0. Specification 7.9 says that the output transaction from the core ($transN\ s_0$) is sent to s_1 ($(transN\ s_0)[s_1]$) if s_1 accepts the request.
To simplify subsequent specifications we introduce Definitions 7.21, 7.22, and 7.23 for a request from s_0 to s_1 at time t, an accept from s_1 to s_0 at time t, and a transfer from s_0 to s_1 at time t. These definitions and the specifications allow the framework to prove Lemma 7.2, which provides a higher level view of how transactions flow between segments.

7.6.2.2 Arbitration Specifications

The specifications for an arbitration scheme are concerned with ensuring that every request is eventually accepted, as shown in Theorem 7.5. The proof of this theorem is described in Section 8.2.1. As described in Section 7.1.2, we use the predicate *holdUntil* to say that every request is accepted. This means that if $req\ (s_0,\ s_1)$ is true at time t_r, then there is

Specification 7.4: *Protocol (Request signals from previous segments)*
$\forall\ s_1,\ t\ .\ reqP\ s_1\ t \iff \exists\ s_0.\ (xReqP\ s_1)[s_0](t)$

Specification 7.5: *Protocol (Accept signals to previous segments)*
$\forall\ s_0,\ s_1,\ t\ .\ (xAccP\ s_1)[s_0](t) \iff (accP\ s_1\ t)\ \&\ (s_0 = prev\ s_1\ t)$

Specification 7.6: *Protocol (Transactions from previous segments)*
$\forall\ s_0,\ s_1,\ t\ .\ (xReqP\ s_1)[s_0](t)\ \&\ (s_0 = prev\ s_1\ t) \implies$
$transP\ s_1\ t = (xTransP\ s_1)[s_0](t)$

Specification 7.7: *Protocol (Request signals to next segments)*
$\forall\ s_0,\ s_1,\ t\ .\ (xReqN\ s_0)[s_1](t) \iff (reqN\ s_0\ t)\ \&\ (s_1 = next\ s_0\ t)$

Specification 7.8: *Protocol (Accept signals from next segments)*
$\forall\ s_0,\ t\ .\ (accN\ s_0\ t) \iff (xAccN\ s_0)\ [next\ s_0\ t](t)$

Specification 7.9: *Protocol (Transactions to next segments)*
$\forall\ s_0,\ s_1,\ t\ .\ (xReqN\ s_0)[s_1](t)\ \&\ (xAccN\ s_0)[s_1](t) \implies$
$(xTransN\ s_0)[s_1](t) = transN\ s_0\ t$

Definition 7.21: *Request between two segments*
$req\ (s_0,\ s_1)\ t \ \hat{=}\ (xReqN\ s_0)[s_1](t)$

Definition 7.22: *Accept between two segments*
$acc\ (s_0,\ s_1)\ t \ \hat{=}\ (xAccP\ s_1)[s_0](t)$

Definition 7.23: *Transfer between two segments*
$xfr\ (s_0,\ s_1)\ t \ \hat{=}$
$(req\ (s_0,\ s_1)\ t)\ \&\ (acc\ (s_0,\ s_1)\ t)\ \&\ (transP\ s_1\ t = transN\ s_0\ t)$

Lemma 7.2: *Correctness of transfers between segments*
$\vdash \forall\ s_0,\ s_1,\ t\ .\ (req\ (s_0,\ s_1)\ t)\ \&\ (acc\ (s_0,\ s_1)\ t) \implies xfr\ (s_0,\ s_1)\ t$

a 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.

Theorem 7.5: *Every request is eventually accepted (Copy of Theorem 8.5)*
$\vdash\ \forall\ s_0,\ s_1,\ t_r\ .\ \exists\ t_a\ .\ holdUntil\ (req\ (s_0,\ s_1))\ t_r\ (acc\ (s_0,\ s_1))\ t_a$

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.

Separating the specifications into these two parts simplifies verifying that an arbitration scheme meets the specifications. In all arbitration schemes that we have found (Section 3.2.2) there is a clear distinction between verifying these properties. For example, in an exclusive arbitration scheme there is at most one active request at a time to each segment. Thus, the active request is the highest priority request. In a static arbitration scheme we prove that the highest priority request is accepted based upon the topology of the pipeline between the current segment and the exit. We prove that every request becomes the highest priority request based on the topology between the entrance and the current segment.

For notational uniformity with *req*, *acc*, and *xfr* we use Definition 7.24 to say that s_1 has selected s_0 as the previous segment and Definition 7.25 to say that s_0 is the highest priority request to s_1.

Definition 7.24: *Select highest priority request (previous segment)*
$prev\ (s_0,\ s_1)\ t\ \hat{=}\ (prev\ s_1\ t = s_0)$

Definition 7.25: *Highest priority request*
$priReq\ (s_0,\ s_1)\ t\ \hat{=}\ (req\ (s_0,\ s_1)\ t)\ \ \&\ \ (prev\ (s_0,\ s_1)\ t)$

Proving that a pipeline meets the protocol specifications can be done by checking that each interface unit meets the specifications and then checking that the connection of signals between the segments is correct. Thus, each unit can be tested in isolation. Reasoning about how transactions flow through the pipeline and contend for segments requires knowing the topology of the pipeline and may require placing constraints on the environment or on latencies through segments in the pipeline. This means that the arbitration specifications are checked for an entire pipeline, rather than each segment in isolation.

Specification 7.10 says that every sustained request to the core of a segment ($reqP\ s_1$) is eventually accepted ($accP\ s_1$). This specification is used in the proof of Lemma 7.3, which says that the highest priority request to a segment is accepted. We wrote the specification in terms of the request to the core, rather than the highest priority request, because the relationship between the highest priority request and the request to the core is dependent upon the interface unit, which must forward the request to the core. Specification 7.10 allows us to ignore the interface circuitry when reasoning about the correctness of an arbitration scheme.

Specification 7.10: *Arbitration (Every sustained request to the core of a segment is eventually accepted)*

$$\forall\ s_1,\ t_r\ .\ \exists\ t_a\ .\ holdCond\ (reqP\ s_1)\ t_r\ (accP\ s_1)\ t_a$$

Lemma 7.3: *Every highest priority request is eventually accepted*

$$\vdash\ \forall\ s_0,\ s_1,\ t_r\ .\ \exists\ t_a\ .\ holdUntil.\ (priReq\ (s_0,\ s_1))\ t_r\ (acc\ (s_0,\ s_1))\ t_a$$

We use *holdUntil* in Lemma 7.3, because it guarantees that requests are both sustained and eventually accepted. Specification 7.10 uses the weaker predicate *holdCond*, so that for a pipeline we only need to prove that highest priority requests are accepted if they are sustained. The separation of the specifications and the use of *holdCond* and *holdUntil* are examples of how the framework decomposes large proof obligations into individual specifications. The reasoning required to combine the specifications together was done in the verification of the framework (*e.g.* in the proof of Lemma 7.3), not in the verification of a pipeline.

In order for the framework to prove that highest priority requests are sustained (which is used in the proof of Lemma 7.3), Specification 7.11 requires that once a request becomes the highest priority request, it maintains its priority until it is accepted. More precisely, Specification 7.11 says that when a segment (s_0) is requesting a second segment (s_1) and s_1 selects s_0 as its previous segment, s_1 continues to select s_0 as long as s_0 requests s_1 and the core of s_1 does not accept a transaction. The requirements placed on the cores of the segments by *CoresOk* (Definition 7.12), the protocol specifications, and Specification 7.11 ensure that highest priority requests are sustained (Lemma 7.4). The framework uses this lemma in the proof of Lemma 7.3, which says that the highest priority request is accepted.

The second part of verifying an arbitration scheme is concerned with showing that every request becomes the highest priority request. The arbitration unit for a segment selects which incoming request has the highest priority. Specification 7.12 says that every segment (s_0) that has a sustained request to s_1 is eventually selected by s_1 as its previous segment ($prev\ (s_0,\ s_1)$).

Specification 7.11: *Arbitration (Highest priority requests maintain their priority)*

$$\begin{array}{l} \forall s_0,\ s_1,\ t\ . \\ \quad (req\ (s_0,\ s_1)\ t)\ \&\ (prev\ (s_0,\ s_1)\ t)\ \&\ \neg(accP\ s_0\ t)\ \& \\ \quad (req\ (s_0,\ s_1)\ (t{+}1))\ \Longrightarrow \\ \qquad prev\ (s_0,\ s_1)\ (t{+}1) \end{array}$$

Lemma 7.4: *Highest priority requests are sustained*

$$\vdash\ \forall s_0,\ s_1\ .\ sustain\ (priReq\ (s_0,\ s_1))\ (acc\ (s_0{,}s_1))$$

7.6.2.3 Control Specification

The specifications for a control scheme describe the required behavior for the functional units *mkInst* and *mkNext*. The functional unit *mkInst* determines the operation that a segment performs. The functional unit *mkNext* determines the next segment to send a transaction to and the data-stationary control information to send with the transaction. 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 pipe is working "correctly." Specification 7.13 says that for every possible input transaction (T_0), there is an output transaction (T_n) such that there is a path from the entry segment to the exit segment where the exit segment produces T_n and selects the bottom environment (*Bot*) as the next segment. We know that there is a path from *Entry* to *Exit* if we can traverse through the combinational representations of the segments. We use *travComb*, as described in Section 7.4, to traverse through the combinational representations of the segments.

Specifications 7.14 and 7.15 say that the implementations of the functional units imply their specifications. Specifications 7.16 and 7.17 say that the specifications of the control units have equal outputs for equal inputs. In Specifications 7.14 and 7.15 we capitalize the variables (*e.g. Inst*), because they are signals and in Specifications 7.16 and 7.17 we do not capitalize the variables because they are values, not signals.

Specification 7.16, Specification 7.17, and the similar requirement in *CoreOk* (Definition 7.13) for the core and its combinational representation allow us to prove that transactions take

Specification 7.12: *Arbitration (Every segment that sustains its request is selected as the previous segment)*

$$\forall\ s_0,\ s_1,\ t_r\ .\ \exists t_a\ .\ holdCond\ (req\ (s_0,\ s_1))\ t_r\ (prev\ (s_0,\ s_1))\ t_a$$

Specification 7.13: *Control (All paths through combinational representation exit pipeline)*

$$\forall T_0 \;.\; \exists T_n \;.\; \mathit{travComb}\ (\mathit{Entry},\ T_0)\ (\mathit{Exit},\ (T_n,\ \mathit{Bot}))$$

the same path through a combinational representation and the implementation. We state this formally for a single step in a path in Lemma 7.5. This lemma says that if a transaction transfers into segment s_0 at time t_0, and the combinational representation of s_0 chooses s_1 as the next segment, then when the transaction exits s_0 (as detected by the matching relation for s_0), the transaction will transfer to s_1. The proof of this lemma is discussed in Section 8.2.2.

Specification 7.14: *Control (Implemention of mkInst implies specification)*

$$\begin{array}{l} \forall\ s,\ \mathit{InstP},\ \mathit{CInstP}\ . \\ \quad (\mathit{mkInst}\ s\ \mathit{InstP}\ \mathit{CInstP}) \implies (\forall t\ .\ \mathit{mkInstSpec}\ s\ (\mathit{InstP}\ t)\ (\mathit{CInstP}\ t))\ ) \end{array}$$

Specification 7.15: *Control (Implemention of mkNext implies specification)*

$$\begin{array}{l} \forall\ s,\ \mathit{InstP},\ \mathit{DataN},\ \mathit{InstN},\ \mathit{Next}\ . \\ \quad (\mathit{mkNext}\ s\ (\mathit{InstP},\ \mathit{DataN})\ (\mathit{InstN},\ \mathit{Next})) \implies \\ \qquad (\forall t\ .\ \mathit{mkNextSpec}\ s\ (\mathit{InstP}\ t,\ \mathit{DataN}\ t)\ (\mathit{InstN}\ t,\ \mathit{Next}\ t)) \end{array}$$

7.6.2.4 Ordering Specification

The purpose of an ordering scheme is to ensure that we can match corresponding input and output transactions. 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. We capture this in Specification 7.18 by saying that there is a transaction that enters the pipeline at t_0 (transfers from *Top* to *Entry*), traverses from *Entry* at time t_0 to *Exit* at time t_n, and exits the pipeline at t_n. We use the relation *travPipe* (Definition 7.20 in Section 7.5) to represent traversing through the implementation of a pipeline. (The relation *travComb* (Definition 7.18 in Section 7.4) is for traversing through the combinational representation of a pipeline).

The relation *isOneToOne* (Definition 7.10 in Section 7.2 and repeated here as Definition 7.26) says that a matching relation defines a one-to-one mapping between the input and output transactions of a pipeline. The definition of *isOneToOne* uses *XfrIn* and *XfrOut* (Definitions 7.27 and 7.28) to denote transferring into and out of a pipeline. When introducing *isOneToOne* in Section 7.2, we did not give the formal definitions of *XfrIn* and *XfrOut*, because they rely on the definition of *xfr* (Definition 7.23), which had not yet been presented.

Specification 7.16: *Control (Specification of mkInst has equal inputs for equal outputs)*

$$\forall\ s,\ instP,\ cInstP,\ cInstP'\ .$$
$$(mkInstSpec\ s\ instP\ cInstP)\ \&\ (mkInstSpec\ s\ instP\ cInstP')\ \Longrightarrow$$
$$(cInstP = cInstP')$$

Specification 7.17: *Control (Specification of mkNext has equal inputs for equal outputs)*

$$\forall\ s,\ instP,\ dataN,\ instN,\ instN',\ next,\ next'\ .$$
$$(mkNextSpec\ s\ (instP,\ dataN)\ (instN,\ next))\ \&$$
$$(mkNextSpec\ s\ (instP,\ dataN)\ (instN',\ next'))\ \Longrightarrow$$
$$(instN = instN')\ \&\ (next = next')$$

Lemma 7.5: *Transactions transfer to the correct next segment (Copy of Lemma 8.17)*

$$\vdash\ \forall s_0,\ s_1,\ t_0,\ t_1,\ T\ .$$
$$match\ s_0\ (t_0,\ t_1)\ \&$$
$$segComb\ s_0\ (transP\ s_0\ t_0)\ (T,\ s_1)\ \Longrightarrow$$
$$xfr\ (s_0,\ s_1)\ t_1$$

Specification 7.18: *Ordering*

$$\forall I,\ t_0,\ O,\ t_n\ .$$
$$Match\ I\ O\ t_0\ t_n\ \ \Longleftrightarrow$$
$$xfr\ (Top,\ Entry)\ t_0\ \&$$
$$travPipe\ (Entry,\ t_0)\ (Exit,\ t_n)\ \&$$
$$xfr\ (Exit,\ Bot)\ t_n$$

Definition 7.26: *One to one mapping between input and output transactions (Copy of Definition 7.10)*

$$isOneToOne\ Match\ I\ O\ \hat{=}$$
$$(\forall t_i\ .\ XfrIn\ I\ t_i\ \ \Longrightarrow\ \exists ! t_o\ .\ Match\ I\ O\ t_i\ t_o)\ \&$$
$$(\forall t_o\ .\ XfrOut\ O\ t_o\ \ \Longrightarrow\ \exists ! t_i\ .\ Match\ I\ O\ t_i\ t_o)\ \&$$
$$(\forall t_i,\ t_o\ .\ Match\ I\ O\ t_i\ t_o\ \ \Longrightarrow\ \ (XfrIn\ I\ t_i)\ \&\ (XfrOut\ O\ t_o))$$

Definition 7.27: *Transfer into a pipeline*

$$XfrIn\ t\ \hat{=}\ xfr\ (Top,\ Entry)\ t$$

Definition 7.28: *Transfer out of a pipeline*

$$\mathit{XfrOut}\ t \ \hat{=}\ \mathit{xfr}\ (\mathit{Exit},\ \mathit{Bot})\ t$$

The framework uses Specification 7.18 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.

As part of the first proof, we use the reverse direction of the *if-and-only-if* in Specification 7.18 to say that if a transaction traverses the pipeline then the matching relation matches its entry and exit times. We then use the control specifications, which say that every path through the pipeline leads to an exit, to prove that every input transaction results in exactly one output transaction.

In the second proof, we use the forward direction of the *if-and-only-if* to say that if the matching relation matches two transactions, then the second results from the first one traversing the pipeline. We then show by induction that traversing a path through the implementation of the pipeline generates transactions that are correct with respect to the combinational representation.

7.7 Conclusion

This chapter began with describing how we model a generic segment in a pipeline. The framework defines behavioral specifications for a set of virtual functional units in a generic segment. The first task in verifying a pipeline using the framework is to define the correspondence between the circuitry in the pipeline and the virtual functional units and then prove that the circuits meet the specifications of the corresponding functional units. There are three additional tasks: we must prove that the core of each segment is correct, the constraints each segment places on its environment are met, and the combinational representation of the pipeline implies the specification.

Proving that the core of each segment is correct is done by using the framework to verify hierarchical pipelines. Using the framework to verify a pipeline is sufficient to prove that the pipeline can be used as the core of a segment. In all pipelines that we have found, proving that the constraints of the segments are met (the second of the three tasks) is very straightforward. The third task is to prove that the combinational representation of the pipeline implies the specification. This proof does not reason about any pipelining concerns. It is purely combinational and can be handled by conventional hardware verification techniques.

Chapter 8

Verification of the Framework

8.1 Introduction

In Chapter 7, we showed how we use a divide and conquer approach to decompose the proof that a pipeline *satisfies* a functional specifications into a set of smaller, separate tasks. This approach is based on two theorems. Theorem 8.1 says that *satisfies* is transitive. This allows us to introduce the combinational representation of a pipeline and separates the pipelining and functionality aspects of verification. Theorem 8.2 simplifies the verification of the pipelining aspects by dividing the proof into a number of independent tasks. The principal task in verifying a pipeline is to show that the pipeline meets the specifications of the four framework parameters. Theorem 8.2 is subsumed by Theorem 8.3, which says that if a pipeline meets the specifications for the framework parameters then it can be used as the core of a segment in hierarchical pipelines. This theorem enables us to progress through the structural hierarchy of a pipeline, using the framework to verify each pipeline and then using that pipeline as the core of a segment in the next level of the hierarchy. The proof of Theorem 8.1 is very easy: we unfold the definition of *satisfies* and apply Modus Ponens. In this chapter we present the proof of Theorem 8.3, which is sufficient to also prove Theorem 8.2.

Theorem 8.1: *Transitivity of satisfies (Copy of Theorem 7.2)*

$$
\begin{array}{l}
\vdash \forall \mathit{PipeRecord},\ \mathit{Comb},\ \mathit{Spec}\ . \\
\qquad \mathit{PipeRecord} \sqsupset \mathit{Comb}\ \& \\
\qquad (\forall T_i,\ T_o\ .\ \mathit{Comb}\ T_i\ T_o \implies \mathit{Spec}\ T_i\ T_o) \implies \\
\qquad\quad \mathit{PipeRecord} \sqsupset \mathit{Spec}
\end{array}
$$

In this first section we give an overview of the proof, briefly discuss the verification of the simple parts of the proof, and introduce several key lemmas that allow us to perform induction over the path that a transaction follows through a pipeline. The proof of Theorem 8.3, just as with the verification of a pipeline, is separated into two major components: pipelining correctness and functionality correctness. Section 8.2 describes the verification of the

Theorem 8.2: *Pipelines that meet framework specifications satisfy their combinational representations (Copy of Theorem 7.3)*

$$
\begin{array}{l}
\vdash \forall \; \mathit{Pipe},\ \mathit{Match},\ \mathit{Constraints},\ \mathit{Comb}\ . \\
\qquad (\mathit{CoresOk}\ \mathit{Pipe})\ \& \\
\qquad (\mathit{ConstraintsOk}\ \mathit{Pipe}\ \mathit{Constraints})\ \& \\
\qquad (\mathit{CombOk}\ \mathit{Pipe}\ \mathit{Comb})\ \& \\
\qquad (\mathit{FworkParamsOk}\ (\mathit{Pipe},\ \mathit{Match},\ \mathit{Constraints}))\ \Longrightarrow \\
\qquad\quad (\mathit{Pipe},\ \mathit{Match},\ \mathit{Constraints}) \sqsupset \mathit{Comb}
\end{array}
$$

Theorem 8.3: *Sufficiency of conditions for a valid core (Copy of Theorem 7.4)*

$$
\begin{array}{l}
\vdash \forall \; \mathit{Pipe},\ \mathit{Match},\ \mathit{Constraints},\ \mathit{Comb}\ . \\
\qquad (\mathit{CoresOk}\ \mathit{Pipe})\ \& \\
\qquad (\mathit{ConstraintsOk}\ \mathit{Pipe}\ \mathit{Constraints})\ \& \\
\qquad (\mathit{CombOk}\ \mathit{Pipe}\ \mathit{Comb})\ \& \\
\qquad (\mathit{FworkParamsOk}\ (\mathit{Pipe},\ \mathit{Match},\ \mathit{Constraints}))\ \Longrightarrow \\
\qquad\quad (\mathit{Pipe},\ \mathit{Match},\ \mathit{Constraints})\ \boxed{\triangleright}\ \mathit{Comb}
\end{array}
$$

pipelining aspects of Theorem 8.3 and Section 8.3 presents the proof of the functionality aspects of correctness.

8.1.1 Outline of Proof of Sufficiency of Conditions for a Valid Core

The first step in proving Theorem 8.3 is to unfold the definition of *CoreOk*, which is defined in terms of *satisfies*. The definition of *satisfies* is written in terms of *PipeliningOk*. We need to prove all of the predicates that result from recursively unfolding these definitions. This process results in the eight predicates shown in Table 8.1. Next to each predicate that is a leaf node is shown the theorem or lemma used to prove the predicate. The predicates at the branch nodes are proved by composing the results their children.

Unfolding *CoreOk* produces five subgoals: 1) the pipeline *satisfies* its combinational representation, 2) the combinational representation is functional, 3) output transactions are held constant while waiting to be accepted, 4) the information fields of matching input and output transactions are equal, and 5) the latency through the pipeline is always non-negative.

The proof that if a pipeline meets the specifications of the framework parameters then it *satisfies* its combinational representation is divided into reasoning about pipelining correctness criteria (*PipeliningOk*) and functional correctness. The functional criteria require that matching input/output transactions are correct with respect to the combinational representation.

The definition of *PipeliningOk* says that every transaction that wants to enter the pipeline is able to do so (*canEnter*) and there is a one-to-one correspondence between input and

Table 8.1: Proof obligations in proof of Theorem 8.3

CoreOk	
Satisfies	
PipeliningOk	
canEnter	
sustain ReqN AccN	Lemma 8.1
holdCond ReqP AccP	Theorem 8.4
isOneToOne	
$\mathit{XfrIn}\ t_i \implies \exists! t_o\ .\ \mathit{Match}\ I\ O\ t_i\ t_o$	Theorem 8.7
$\mathit{XfrOut}\ t_o \implies \exists! t_i\ .\ \mathit{Match}\ I\ O\ t_i\ t_o$	Theorem 8.8
$\mathit{Match}\ I\ O\ t_i\ t_o \implies \mathit{XfrIn}\ t_i\ \&\ \mathit{XfrOut}\ t_o$	Specification 8.5
$\mathit{Match}\ I\ O\ t_i\ t_o \implies \mathit{Spec}\ (\ldots\ (I\ t_i))\ (\ldots\ (O\ t_o))$	Theorem 8.10
$(\mathit{Comb}\ T_i\ T_o)\ \&\ (\mathit{Comb}\ T_i\ T_o') \implies T_o = T_o'$	Lemma 8.2
Hold *TransN* constant while waiting for *AccN*	Lemma 8.3
$\mathit{Match}\ I\ O\ t_i\ t_o \implies (\mathit{getInfo}\ (O\ t_o)) = (\mathit{getInfo}\ (I\ t_i))$	Lemma 8.4
$\mathit{Match}\ I\ O\ t_i\ t_o \implies (t_i \leq t_o)$	Lemma 8.5

output transactions (*isOneToOne*). The first clause under *canEnter* is that the pipeline sustains its requests to the environment. The second is that the pipeline accepts every sustained request. Together, these two conditions are sufficient to guarantee that in a valid environment (*envOk*, Definition 7.7) all transactions that wish to enter the pipeline are able to do so. The first two clauses under *isOneToOne* are to prove that every input transaction matches exactly one output (Theorem 8.7) and that every output transaction matches exactly one input (Theorem 8.8). The third clause says that the matching relation only matches times when transactions enter and exit the pipeline.

There are ten proof goals in Table 8.1. Of these, four are proved using theorems, five using lemmas, and one directly follows from the specification for ordering schemes. Theorem 8.4 guarantees that every sustained request is accepted. It is proved in Section 8.2.1. Theorems 8.7 and 8.8 say that every input transaction produces exactly one output and every output results from exactly one input. Theorem 8.7 is the more difficult of the two directions; its proof (Section 8.2.3) relies on Theorem 8.4 and a lemma that says that transactions make progress following the correct path through the pipeline (Section 8.2.2). The proof of Theorem 8.8 is described in Section 8.2.4. The fourth theorem says that any pair of matching input/output transactions are correct with respect to the combinational representation (Theorem 8.10). The remaining six criteria follow from lemmas proved using the protocol specifications, ordering specification, and the definitions of *CoresOk* and *CombOk*. Their proofs are discussed in Section 8.1.2.

The definition of *FworkParamsOk* (Definition 8.1) has as antecedents that the core of each segment meets the requirements in *CoreOk* and the constraints of the segments are met.

Definition 8.1: *Specifications for framework parameters are satisfied (Copy of Definition 7.16)*

$$
\begin{array}{l}
\quad FworkParamsOk\ (Pipe,\ Match,\ Constraints) \mathrel{\hat{=}} \\
\qquad CoresOk\ Pipe\ \& \\
\qquad ConstraintsOk\ Pipe\ Constraints \implies \\
\qquad\quad \forall\ I,\ O\ .\ (Pipe\ I\ O)\ \&\ (Constraints\ I\ O)\ \&\ (envOk\ I\ O) \implies \\
\qquad\qquad \begin{array}{lll}
ProtocolOk & Pipe & \& \\
ArbitrationOk & Pipe & \& \\
ControlOk & Pipe & \& \\
OrderingOk & Pipe\ Match &
\end{array}
\end{array}
$$

In order to use the specifications in the proof of Theorem 8.3 we need to prove that the antecedents to *FworkParamsOk* hold. They do, because the antecedents of Theorem 8.3 include these same two conditions. Another antecedent in *FworkParamsOk* is that the environment must be valid and meet the constraints of the pipeline. This antecedent is solved when we unfold the definitions of *CoreOk* and *satisfies* in Theorem 8.3.

Subsequent theorems and proofs in this chapter are implicitly stated in a *context* in which the cores of the segments are valid (*CoresOk*), the constraints of the segments are met (*ConstraintsOk*), the combinational representation of the pipeline is valid (*CombOk*), and the framework specifications are true. The context also includes declarations for the signals and virtual functional units of the segments (*e.g.* $req\ (s_0,\ s_1)$ is defined). This means that the statements of the subsequent theorems do not include *CoresOk*, *ConstraintsOk*, etc as hypotheses and do not declare *req*, *core*, etc.

8.1.2 Simple Goals in Proof of Sufficiency of Conditions for a Valid Core

In Section 8.1.1 we began the proof of Theorem 8.3 by recursively unfolding definitions to reveal eight subgoals. Five of these subgoals are very simple; we outline their proofs in this section. The four significant subgoals are discussed in Sections 8.2 and 8.3.

The first of the six simple proof goals is that requests from the pipeline are sustained until they are accepted. Lemma 8.1 says that requests from all segments are sustained until they are accepted. The request from the pipeline to the environment is just a request from the exit segment (*Exit*) to the bottom environment (*Bot*). The proof of Lemma 8.1 follows from the requirements in *CoresOk* that the cores of segments sustain their requests and the contents of output transactions do not change while they are waiting for a request to be accepted. We require that the contents of transactions do not change while waiting to be accepted, because the identity of the segment that is selected as the next segment is dependent upon the control information and data in the transaction. Allowing the contents of the transaction to change could cause a different segment to be selected as the next segment, resulting in the request to the original segment to not be sustained.

Lemma 8.1: *Requests are sustained*
$\vdash \forall s_0,\ s_1\ .\ \mathit{sustain}\ (\mathit{req}\ (s_0,\ s_1))\ (\mathit{acc}\ (s_0, s_1))$

The second goal is that if *Match* returns true for two times t_0 and t_n, then a transfer into the pipeline occurred at t_0 and a transfer out occurred at t_n. This is proved using the ordering specification (Specification 8.1), which says that if *Match* returns true for t_0 and t_n then a transaction enters at t_0, traverses the pipeline, and exits at t_n.

Specification 8.1: *Ordering (Copy of Specification 7.18)*

$$
\begin{array}{l}
\forall I,\ t_0,\ O,\ t_n\ . \\
\quad \mathit{Match}\ I\ O\ t_0\ t_n \iff \\
\quad\quad \mathit{xfr}\ (\mathit{Top},\ \mathit{Entry})\ t_0\ \&\ \\
\quad\quad \mathit{travPipe}\ (\mathit{Entry},\ t_0)\ (\mathit{Exit},\ t_n)\ \&\ \\
\quad\quad \mathit{xfr}\ (\mathit{Exit},\ \mathit{Bot})\ t_n
\end{array}
$$

The third goal is that *Comb* is functional, that is, it has equal outputs for equal inputs. The goal is solved using Lemma 8.2. The proof follows from the definition of *CoresOk*, which says that the cores of the segments are functional; the control specifications that say the specifications of the control circuitry for each segment are functional; and *CombOk*, which requires that *Comb* is equivalent to traversing the combinational representations of the segments.

Lemma 8.2: *Combinational representations are functional*
$\vdash \forall T_i,\ T_o,\ T_o'\ .\ (\mathit{Comb}\ T_i\ T_o)\ \&\ (\mathit{Comb}\ T_i\ T_o') \implies (T_o = T_o')$

The fourth goal is that the output transaction from the pipeline is held constant while it is waiting for its request to be accepted (Lemma 8.3). As with the first of the four goals (requests from the pipeline are sustained) this requirement is imposed on the segments by *CoresOk* and is true for a pipeline because the output transaction is just the output transaction of the exit segment.

The fifth and sixth simple proof goals say that the information field in a transaction does not change as it traverses the pipeline (Lemma 8.4) and the latency through the pipeline is always non-negative (Lemma 8.5). These goals come from requirements in *CoreOk* and are proved using lemmas derived from the fact that the cores of the segments in the pipeline meet the requirements in *CoreOk*.

8.1.3 Traversing a Pipeline and Induction

The key to the verification of the framework is the notion of traversing a path between two segments. We use two relations: *travComb* (Definition 8.2) for combinational repre-

Lemma 8.3: *Hold output transactions constant while waiting for an accept*
$\vdash \forall s_0,\ s_1,\ t\ .$
$$(req\ (s_0,\ s_1)\ t)\ \&\ \neg(acc\ (s_0,\ s_1)\ t) \implies transN\ s_0\ (t{+}1) = transN\ s_0\ t$$

Lemma 8.4: *Matching transactions have equal info fields*
$\vdash \forall\ t_i,\ t_o\ .\ Match\ I\ O\ t_i\ t_o \implies (getInfo\ (O\ t_o)) = (getInfo\ (I\ t_i))$

sentations and *travPipe* (Definition 8.3) for implementations of pipelines. A transaction traverses from the first segment (s_0) in a path to the last segment (s_n) if it enters s_0, follows a path from s_0 to s_n, and exits s_n. The relations *travPipe* and *travComb* are defined inductively. In the base case the first and last segments in the path are the same ($s_0{=}s_n$) and the transaction uses the segment and completes the traversal. In the inductive case, there is some intermediate segment (s_x) such that the transaction uses s_0, transfers from s_0 to s_x, and then traverses from s_x to s_n.

If we have a relation $\mathcal{R}$ defined by a set of positive rules* and we intend $\mathcal{R}$ to be the smallest relation closed under the rules, which is the standard interpretation, then we can inductively prove that $\mathcal{R}$ implies a property $\mathcal{P}$ by replacing each occurrence of $\mathcal{R}$ in the rules with $\mathcal{P}$ and proving that each of these new rules implies $\mathcal{P}$ [Mos74, Bar77]. Camelleri and Melham have automated this process in HOL [CM92]. This chapter includes two proofs that rely on this technique. In the proof that every input transaction produces one output transaction (Theorem 8.7, Section 8.2.3), we perform induction on *termComb* (Definition 8.4). In Section 8.3 we perform induction on *travPipe* to prove that every output is correct with respect to the combinational representation (Theorem 8.10). The proof that every output transaction results from exactly one input (Theorem 8.8) performs induction on the length of paths through a pipeline. It relies on the fact that *travPipe* is defined inductively, but does not perform induction on *travPipe*.

In Sections 7.4 and 7.5 we defined *travComb* and *travPipe* inductively. Applying the technique described in the previous paragraph to *travComb* and *travPipe* results in induction lemmas for the two relations (Lemmas 8.6 and 8.7).

Two important properties that we reason about are termination of transactions in combinational representations and implementations. We define termination in a combinational representation (*termComb*, Definition 8.4) and in an implementation (*termPipe*, Definition 8.7 in Section 8.2.3) in terms of *travComb* and *travPipe*. A transaction terminates if it traverses a path from its current location to the exit segment and then transfers from the

*Positive rules define when the relation is true, as opposed to defining when it is false

Lemma 8.5: *Latency through a pipeline is always non-negative*
$\vdash \forall\ t_i,\ t_o\ .\ Match\ I\ O\ t_i\ t_o \implies t_i \le t_o$

Definition 8.2: *Traverse through a combinational representation (Copy of Definition 7.18)*

$$
\begin{array}{l}
travComb\ (s_0,\ T_0)\ (s_n,\ (T_n,\ s'))\ \hat{=} \\
\quad (s_0 = s_n)\ \&\ (segComb\ s_0\ T_0\ (T_n,\ s')) \\
\quad \vee \\
\quad \exists s_x,\ T_x\ . \\
\quad\quad segComb\ s_0\ T_0\ (T_x,\ s_x)\ \& \\
\quad\quad travComb\ (s_x,\ T_x)\ (s_n,\ (T_n,\ s'))
\end{array}
$$

Definition 8.3: *Traverse through the implementation of a pipeline (Copy of Definition 7.20)*

$$
\begin{array}{l}
travPipe\ (s_0,\ t_0)\ (s_n,\ t_n)\ \hat{=} \\
\quad (s_0 = s_n)\ \&\ (match\ s_0\ (t_0,\ t_n)) \\
\quad \vee \\
\quad \exists\ s_x,\ t_x\ . \\
\quad\quad match\ s_0\ (t_0,\ t_x)\ \& \\
\quad\quad xfr\ (s_0,\ s_x)\ t_x\ \& \\
\quad\quad travPipe\ (s_x,\ t_x)\ (s_n,\ t_n)
\end{array}
$$

Lemma 8.6: *Induction lemma for traversing a combinational representation*

$$
\begin{array}{l}
\vdash\ \forall\ \mathcal{P}\ . \\
\quad (\forall\ s_0,\ s_n,\ s',\ T_0,\ T_n\ . \\
\quad\quad s_0{=}s_n\ \ \&\ \ (segComb\ s_0\ T_0\ (T_n{,}s'))\ \ \Longrightarrow \\
\quad\quad\quad \mathcal{P}\ (s_0,\ T_0)\ (s_n,\ (T_n,\ s')) \\
\quad)\ \& \\
\quad (\forall\ s_0,\ s_n,\ s',\ T_0,\ T_n\ . \\
\quad\quad (\exists s_x,\ T_x\ . \\
\quad\quad\quad segComb\ s_0\ T_0\ (T_x,\ s_x)\ \& \\
\quad\quad\quad \mathcal{P}\ (s_x,\ T_x)\ (s_n,\ (T_n,\ s')) \\
\quad\quad)\ \Longrightarrow \\
\quad\quad\quad \mathcal{P}\ (s_0,\ T_0)\ (s_n,\ (T_n,\ s')) \\
\quad)\ \Longrightarrow \\
\quad\quad \forall\ s_0,\ s_n,\ s',\ T_0,\ T_n\ . \\
\quad\quad\quad travComb\ (s_0,\ T_0)\ (s_n,\ (T_n,\ s'))\ \ \Longrightarrow\ \ \mathcal{P}\ (s_0,\ T_0)\ (s_n,\ (T_n,\ s'))
\end{array}
$$

Lemma 8.7: *Induction lemma for traversing an implementation*

$$
\begin{array}{l}
\vdash \forall \mathcal{P}. \\
\quad (\forall\, s_0,\ t_0,\ s_n,\ t_n\ . \\
\qquad (s_0 = s_n)\ \&\ (match\ s_0\ t_0\ t_n) \implies \\
\qquad\quad \mathcal{P}\ (s_0,\ t_0)\ (s_n,\ t_n) \\
\quad)\ \& \\
\quad (\forall\, s_0,\ t_0,\ s_n,\ t_n\ . \\
\qquad (\exists\, s_x,\ t_x\ . \\
\qquad\quad match\ s_0\ t_0\ t_x\ \& \\
\qquad\quad xfr\ (s_0,\ s_x)\ t_x\ \& \\
\qquad\quad \mathcal{P}\ (s_x,\ t_x)\ (s_n,\ t_n) \\
\qquad) \implies \\
\qquad\quad \mathcal{P}\ (s_0,\ t_0)\ (s_n,\ t_n) \\
\quad) \implies \\
\qquad \forall\, s_0,\ t_0,\ s_n,\ t_n\ . \\
\qquad\quad travPipe\ (s_0,\ t_0)\ (s_n,\ t_n) \implies \mathcal{P}\ (s_0,\ t_0)\ (s_n,\ t_n)
\end{array}
$$

exit segment to the environment.
In *termComb*, a transaction T_0 terminates from segment s_0 if there is an output transaction T_n such that traversing the combinational representations of the segments leads to the exit segment producing T_n and selecting *Bot* (the bottom environment) as the next segment. Because *termComb* is defined in terms of *travComb*, we use Lemma 8.6 to prove the induction lemma for *termComb* (Lemma 8.8).

Definition 8.4: *Termination for combinational representation*

$$
\begin{array}{l}
termComb\ (s_0,\ T_0)\ \hat{=} \\
\quad \exists\, T_n\ .\ travComb\ (s_0,\ T_0)\ (Exit,\ (T_n,\ Bot))
\end{array}
$$

8.2 Proof of Pipelining Correctness

In Section 8.1.1 we began the proof that pipelines verified with the framework can be used as cores of segments in hierarchical pipelines. We unfolded the definitions of *CoreOk*, *satisfies*, *PipeliningOk*, *canEnter*, and *isOneToOne* to produce eight subgoals. Four subgoals were relatively simple to prove and four required significant effort. Three of the four are related to pipelining correctness and one to functionality correctness. This section presents the proofs of the three theorems that show pipelines verified with the framework meet the pipelining aspects of correctness. Section 8.3 presents the proof that the functionality aspects of correctness are met. We begin in Section 8.2.1 by showing that every sustained request in the pipeline is accepted (Theorem 8.4). We use this result to prove that all

Lemma 8.8: *Induction lemma for termination in a combinational representation*

$$
\begin{array}{l}
\vdash \forall \mathcal{P},\ s_0,\ T_0\ . \\
\quad (s_0{=}\mathit{Exit}\ \ \&\ \ (\exists\ T_n\ .\ \mathit{segComb}\ s_0\ T_0\ (T_n,\ \mathit{Bot}))\ \ \Longrightarrow \\
\qquad \mathcal{P}\ (s_0,\ T_0) \\
\quad)\ \& \\
\quad (\quad (\exists s_x,\ T_x\ . \\
\qquad\quad \mathit{segComb}\ s_0\ T_0\ (T_x,\ s_x)\ \& \\
\qquad\quad \mathcal{P}\ (s_x,\ T_x) \\
\qquad)\ \Longrightarrow \\
\qquad\quad \mathcal{P}\ (s_0,\ T_0) \\
\quad)\ \Longrightarrow \\
\qquad \mathit{termComb}\ (s_0,\ T_0)\ \ \Longrightarrow\ \ \mathcal{P}\ (s_0,\ T_0)
\end{array}
$$

requests are accepted and that the environment of each segment in the pipeline is valid. Section 8.2.2 outlines the proof that transactions make progress traversing a path through the pipeline and follow the correct path. The third proof (Section 8.2.3) uses the lemma that transactions make progress along the correct path to prove that every input transaction produces exactly one output (Theorem 8.7). The pipelining correctness criteria also require that every output results from exactly one input (Theorem 8.8). This proof is somewhat analogous to the forward direction (every input produces one output), but is simpler. We describe the unique aspects of the proof in Section 8.2.4.

8.2.1 Every Sustained Request is Eventually Accepted

In this section we prove that all sustained requests to segments within the pipeline are eventually accepted (Theorem 8.4). The theorem uses *holdCond* to capture the notion of a sustained request eventually being accepted. Theorem 8.4 says that if there is a request from segment s_0 to s_1 at time t_r ($\mathit{req}(s_0,\ s_1)\ t_r$), then there is a time t_a such that if the request is sustained until t_a then s_1 will accept the request from s_0 at t_a ($\mathit{acc}\ (s_0,\ s_1)\ t_a$). At the end of this section we use this theorem to prove two closely related theorems. Theorem 8.5 says that *all* requests are accepted and Theorem 8.6 says that the environment of the core of each segment in the pipeline meets the requirements in *envOk*.

Theorem 8.4: *Every sustained request is accepted*

$$
\begin{array}{l}
\vdash \forall\ s_0,\ s_1,\ t_r\ . \\
\quad \exists\ t_a\ .\ \mathit{holdCond}\ (\mathit{req}\ (s_0,\ s_1))\ t_r\ (\mathit{acc}\ (s_0,\ s_1))\ t_a
\end{array}
$$

We prove Theorem 8.4 using three lemmas: first we show that every sustained request eventually becomes the highest priority request to a segment (Lemma 8.9); second we prove that if the highest priority request is sustained, then it is eventually accepted (Lemma 8.10);

finally we make use of the fact that *holdCond* is transitive (Lemma 8.11) to combine these results together and prove that every sustained request is accepted.

Lemma 8.9: *Every segment with a sustained request is selected as the previous segment*

$$\vdash \forall\ s_0,\ s_1,\ t_r\ .$$
$$\exists\ t_p\ .\ holdCond\ (req\ (s_0,\ s_1))\ t_r\ (priReq\ (s_0,\ s_1))\ t_p$$

Lemma 8.10: *Every sustained highest-priority request is eventually accepted*

$$\vdash \forall\ s_0,\ s_1,\ t_p\ .$$
$$\exists\ t_a\ .\ holdCond\ (priReq\ (s_0,\ s_1))\ t_p\ (acc\ (s_0,\ s_1))\ t_a$$

Definition 8.5 says that a request from a segment (s_0) to another segment (s_1) is the highest priority request to s_1 at time t if s_0 requests s_1 and the arbitration unit for s_1 selects s_0 as the previous segment.

Definition 8.5: *Highest priority request (Copy of Definition 7.25)*

$$priReq\ (s_0,\ s_1)\ t\ \hat{=}\ (req\ (s_0,\ s_1)\ t)\ \ \&\ \ (prev\ (s_0,\ s_1)\ t)$$

Lemma 8.9 is proved using Specification 8.2 and the definition of *priReq*. Specification 8.2 says that if segment s_0 has a sustained request to segment s_1, then s_1 eventually selects s_0 as its previous segment. The proof of Lemma 8.10 uses Specification 8.3, the definition of *priReq*, and the protocol specifications that relate the request between s_0 and s_1 ($acc\ (s_0,\ s_1)$) to the request to the core of s_1 ($accP\ s_1$). Specification 8.3 says that every sustained request to the core of a segment ($reqP\ s_1$) must be eventually accepted ($accP\ s_1$). The proof of Theorem 8.4 is shown in Figure 8.1. In the first two steps we use Lemmas 8.9 and 8.10 to add the hypotheses that every sustained request becomes the highest priority request and every sustained highest priority request is eventually accepted. We then instantiate the existentially quantified variable t_a in the conclusion with the t_a introduced by Lemma 8.10.

Lemma 8.11 says that *holdCond* is transitive for predicates that meet two criteria. To use Lemma 8.11 in this proof we must show that highest-priority requests are sustained until they are accepted (Lemma 8.12) and that if s_0 requests s_1 and s_1 accepts s_0, then s_0 has the highest-priority request to s_1 (Lemma 8.13). This completes the proof of Theorem 8.4 — every sustained request is eventually accepted.

Lemma 8.12 is proved using Lemma 8.1 (all requests are sustained) and Specification 8.4 (once s_1 has selected s_0 as the previous segment, it sustains the selection until the core of s_1 accepts a request). The proof of Lemma 8.13 follows directly from unfolding the definition of *priReq* and the protocol specifications.

Specification 8.2: *Arbitration (Every segment that sustains its request is selected as the previous segment) (Copy of Specification 7.12)*

$\forall\ s_0,\ s_1,\ t_r\ .\ \exists t_a\ .\ holdCond\ (req\ (s_0,\ s_1))\ t_r\ (prev\ (s_0,\ s_1))\ t_a$

Specification 8.3: *Arbitration (Every sustained request to the core of a segment is eventually accepted) (Copy of Specification 7.10)*

$\forall\ s_1,\ t_r\ .\ \exists\ t_a\ .\ holdCond\ (reqP\ s_1)\ t_r\ (accP\ s_1)\ t_a$

Theorem 8.4 says that all sustained requests are accepted. Our complete correctness criteria is that *all* requests are accepted. This is stated in Theorem 8.5, which uses *holdUntil*, rather than *holdCond*. Throughout the framework we maintain separate obligations for requests (that they are sustained) and accepts (that they are eventually true for sustained requests). Because of this, Theorem 8.5 is not used in the proof of Theorem 8.3, which says that any pipeline that meets the framework parameters is a valid core. Instead, Theorem 8.5 acts as a check that Theorem 8.3 accurately represents our intention of what it means for a pipeline to be correct. The proof of Theorem 8.5 uses Theorem 8.4 and Lemma 8.1, which says that all requests are sustained until they are accepted. The proof of Lemma 8.1 is outlined in Section 8.1.2.

Theorem 8.6 is a second theorem that is closely based on Theorem 8.4. It says that the environment of the core of each segment is valid and is proved directly from Theorem 8.4, Lemma 8.1 (all requests are sustained until accepted) and the protocol specifications. Theorem 8.6 uses the input and output bundles of signals for a segment (*bundleP* and *bundleN*, Section 7.1.1), which are analogous to the input and output streams (I and O) for pipelines.

8.2.2 Transactions Follow the Correct Path

In the previous section (Section 8.2.1), we proved that every sustained request is eventually accepted (Theorem 8.4) and used this in the proof that the environment of the core of each segment is valid (Theorem 8.6). In this section we use Theorem 8.6 to prove Lemma 8.14,

Lemma 8.11: *Transitivity of holdCond*

$$
\begin{array}{l}
\vdash\ \forall\ \mathcal{P}_0,\ \mathcal{P}_1,\ \mathcal{P}_2\ . \\
\quad\quad sustain\ \mathcal{P}_1\ \mathcal{P}_2\ \& \\
\quad\quad (\forall\ t\ .\ \mathcal{P}_0\ t\ \Longrightarrow\ \mathcal{P}_2\ t\ \Longrightarrow\ \mathcal{P}_1\ t)\ \Longrightarrow \\
\quad\quad\quad \forall t_0,\ t_1,\ t_2 \\
\quad\quad\quad\quad holdCond\ \mathcal{P}_0\ t_0\ \mathcal{P}_1\ t_1\ \& \\
\quad\quad\quad\quad holdCond\ \mathcal{P}_1\ t_1\ \mathcal{P}_2\ t_2\ \Longrightarrow \\
\quad\quad\quad\quad\quad holdCond\ \mathcal{P}_0\ t_0\ \mathcal{P}_2\ t_2
\end{array}
$$

$\vdash$ $\exists t_a$. *holdCond* (*req* $(s_0,\ s_1)$) t_r (*acc* $(s_0,\ s_1)$) t_a

BY `Lemma 8.9 (*` *holdCond req priReq* `*)`

- *holdCond* (*req* $(s_0,\ s_1)$) t_r (*priReq* $(s_0,\ s_1)$) t_p

$\vdash$ $\exists t_a$. *holdCond* (*req* $(s_0,\ s_1)$) t_r (*acc* $(s_0,\ s_1)$) t_a

BY `Lemma 8.10 (*` *holdCond priReq acc* `*)`

- *holdCond* (*req* $(s_0,\ s_1)$) t_r (*priReq* $(s_0,\ s_1)$) t_p
- *holdCond* (*priReq* $(s_0,\ s_1)$) t_p (*acc* $(s_0,\ s_1)$) t_a

$\vdash$ $\exists t_a$. *holdCond* (*req* $(s_0,\ s_1)$) t_r (*acc* $(s_0,\ s_1)$) t_a

BY `Instantiate concl with` $t_a{=}t_a$
`THEN Lemma 8.11 (*` Transitivity of *holdCond* `*)`

$\vdash$ *sustain* (*priReq* $(s_0,\ s_1)$) (*acc* $(s_0,\ s_1)$)

BY `Lemma 8.12`

■

$\vdash$ $\forall\ t$.
(*req* $(s_0,\ s_1)\ t$) $\Longrightarrow$
(*acc* $(s_0,\ s_1)\ t$) $\Longrightarrow$ (*priReq* $(s_0,\ s_1)\ t$)

BY `Lemma 8.13`

■

- *holdCond* (*req* $(s_0,\ s_1)$) t_r (*acc* $(s_0,\ s_1)$) t_a

$\vdash$ *holdCond* (*req* $(s_0,\ s_1)$) t_r (*acc* $(s_0,\ s_1)$) t_a

■

Figure 8.1: Theorem 8.4 — every sustained request is accepted

---

Lemma 8.12: *Highest-priority requests are sustained*
$\vdash$ $\forall s_0,\ s_1$. *sustain* (*priReq* $(s_0,\ s_1)$) (*acc* (s_0,s_1))

Lemma 8.13: *Accept highest priority request*
$\vdash \forall\ s_0,\ s_1\ .\ \forall\ t\ .\ (req\ (s_0,\ s_1)\ t)\ \&\ (acc\ (s_0,\ s_1)\ t) \implies (priReq\ (s_0,\ s_1)\ t)$

Specification 8.4: *Arbitration (Highest priority requests maintain their priority) (Copy of Specification 7.11)*

$$\begin{array}{l} \forall s_0,\ s_1,\ t\ . \\ \quad (req\ (s_0,\ s_1)\ t)\ \&\ (prev\ (s_0,\ s_1)\ t)\ \&\ \neg(accP\ s_0\ t)\ \& \\ \quad (req\ (s_0,\ s_1)\ (t{+}1)) \implies \\ \qquad prev\ (s_0,\ s_1)\ (t{+}1) \end{array}$$

which says that whenever a transaction enters a segment ($xfrIn\ s_0\ t_0$) there is exactly one time (t_1) when the transaction exits s_0 (as detected by *match* s_0) and when the transaction exits s_0 it transfers to the correct segment. This guarantees both that transactions can make progress traversing through the pipeline and that they follow the correct path. The *correct* next segment is determined by the combinational representation ($segComb\ s_0$). The combinational representation takes a transaction as an input ($transP\ s_0\ t_0$ in Lemma 8.14) and produces an output transaction (T) and the name of the next segment to go to (s_1). Lemma 8.14 is used in the proof that every input transaction produces exactly one output transaction (Theorem 8.7).

Lemma 8.14 is proved using Theorem 8.6 and Lemmas 8.15 and 8.17 (Figure 8.2). Lemma 8.15 says that if the environment of the core of a segment (s_0) is valid, then the matching relation for s_0 matches each input transaction with exactly one output. After applying Theorem 8.6 and Lemma 8.15 we know that the transaction will exit s_0.

We need to prove that there is exactly one time (t_1) such that the transaction that enters s_0 at t_0 exits from s_0 at t_1 and transfers to s_1. That is, we must prove both existence and uniqueness for t_1. To prove uniqueness and existence for multiple conjuncts, we can prove existence for all of the conjuncts together and uniqueness for one conjunct. For example, if we have two conjuncts ($\mathcal{P}_0$ and $\mathcal{P}_1$) where there is some x for which both $\mathcal{P}_0$ and $\mathcal{P}_1$ are true and there is only one x for which $\mathcal{P}_0$ is true, then there is exactly one x for which both $\mathcal{P}_0$ and $\mathcal{P}_1$ are true (Lemma 8.16). From the hypotheses we know that there is a unique t_1 for which the *match* is true. We divide the goal into a existence proof for *match* and *xfr* and a uniqueness proof for *match*. The existence subgoal guarantees that transactions traverse the *correct* path through the pipeline. To solve it, we use Lemma 8.17, which relates the segment that the combinational representation chooses as the next segment and the segment to which the transaction transfers. The uniqueness subgoal is solved from the

Theorem 8.5: *Every request is eventually accepted*
$\vdash \forall\ s_0,\ s_1,\ t_r\ .\ \exists\ t_a\ .\ holdUntil\ (req\ (s_0,\ s_1))\ t_r\ (acc\ (s_0,\ s_1))\ t_a$

Theorem 8.6: *The environment of the core of each segment is correct*
$\vdash \forall\ s\ .\ envOk\ (bundleP\ s)\ (bundleN\ s)$

Lemma 8.14: *Correctness of the path between segments*

$$\begin{array}{l} \vdash \forall s_0,\ s_1,\ t_0,\ T\ . \\ \qquad xfrIn\ s_0\ t_0\ \& \\ \qquad segComb\ s_0\ (transP\ s_0\ t_0)\ (T,\ s_1) \implies \\ \qquad\quad \exists!\ t_1\ .\ match\ s_0\ t_0\ t_1\ \& \\ \qquad\qquad\quad xfr\ (s_0,\ s_1)\ t_1 \end{array}$$

hypotheses.
Lemma 8.15 is derived from the requirement in *CoreOk* (Definition 7.13) that the segments *satisfy* their combinational representations, which means that their matching relations define a one-to-one mapping between input and output transactions. The clause in *CoreOk* that says that the cores of the segments *satisfy* their combinational representations has as an antecedent that the environment of each core is valid. Theorem 8.6, which was proved in Section 8.2.1, says that the environment of each core is valid.
The proof of Lemma 8.17 appears in Appendix D.5; it relies on a lemma that says the implementations of the segments imply the combinational representations and one that says the combinational representations are functional. These lemmas are proved from the definition of *CoreOk* and the control specifications.
The lemma proved in this section (Lemma 8.14) is used in the proof that every input transaction produces exactly one output (Section 8.2.3), which is a pipelining correctness criteria. It may be surprising then, that Lemma 8.14 says both that transactions make progress and follow the correct path, because following the correct path seems like a functionality criteria. If the transaction does not follow the correct path through the pipeline then it is possible that the transaction will loop an infinite number of times through a set of segments in the pipeline. The protocol and arbitration specifications ensure that the transaction is always able to transfer to the next segment, but without the control specifications there is no requirement that the transaction ever attempts to exit the pipeline.

8.2.3 All Input Transactions Traverse the Pipeline

In this section we prove that the matching relation for the pipeline matches each input transaction with exactly one output (Theorem 8.7). In Figure 8.3, we begin the proof with

Definition 8.6: *Transfer into a segment*
$xfrIn\ s_1\ t\ \hat{=}\ \exists\ s_0\ .\ xfr\ (s_0,\ s_1)\ t$

- $xfrIn\ s_0\ t_0$
- $segComb\ s_0\ (transP\ s_0\ t_0)\ (T,\ s_1)$

$\vdash\ \exists!\ t_1\ .\ (match\ s_0\ t_0\ t_1)\ \ \&\ \ (xfr\ (s_0,\ s_1)\ t_1)$

BY `Theorem 8.6 (*` $envOk\ (bundleP\ s_0)\ (bundleN\ s_0)$ `*)`

- $xfrIn\ s_0\ t_0$
- $segComb\ s_0\ (transP\ s_0\ t_0)\ (T,\ s_1)$
- $envOk\ (bundleP\ s_0)\ (bundleN\ s_0)$

$\vdash\ \exists!\ t_1\ .\ (match\ s_0\ t_0\ t_1)\ \ \&\ \ (xfr\ (s_0,\ s_1)\ t_1)$

BY `Lemma 8.15`
`(*` $envOk\ (\ldots)\ \Longrightarrow\ xfrIn\ s_0\ t_0\ \Longrightarrow\ \exists!\ t_1\ .\ match\ s_0\ t_0\ t_1$ `*)`

- $segComb\ s_0\ (transP\ s_0\ t_0)\ (T,\ s_1)$
- $\exists!\ t_1\ .\ match\ s_0\ t_0\ t_1$

$\vdash\ \exists!\ t_1\ .\ (match\ s_0\ t_0\ t_1)\ \ \&\ \ (xfr\ (s_0,\ s_1)\ t_1)$

BY `Lemma 8.16 (*` split goal for unique existential quantifier `*)`

- $match\ s_0\ t_0\ t_1$
- $segComb\ s_0\ (transP\ s_0\ t_0)\ (T,\ s_1)$

$\vdash\ \exists\ t_1\ .\ (match\ s_0\ t_0\ t_1)\ \&\ (xfr\ (s_0,\ s_1)\ t_1)$

BY `Instantiate concl with` $t_1{=}t_1$
`THEN Lemma 8.17`
`(*` $(match\ \ldots)\ \&\ (segComb\ \ldots)\ \Longrightarrow\ (xfr\ \ldots)$ `*)`

- $match\ s_0\ t_0\ t_1$
- $xfr\ (s_0,\ s_1)\ t_1$

$\vdash\ (match\ s_0\ t_0\ t_1)\ \&\ (xfr\ (s_0,\ s_1)\ t_1)$

■

- $\exists!\ t_1\ .\ match\ s_0\ t_0\ t_1$

$\vdash\ \exists!\ t_1\ .\ match\ s_0\ t_0\ t_1$

■

Figure 8.2: Proof of Lemma 8.14 — progress and correctness of paths

Lemma 8.15: *Each input transaction to a segment produces one output*

$$\vdash \forall\, s_0\ .\ envOk\ (bundleP\ s_0)\ (bundleN\ s_0) \implies$$
$$\forall\, t_0\ .\ xfrIn\ s_0\ t_0 \implies \exists!\ t_1\ .\ match\ s_0\ t_0\ t_1$$

Lemma 8.16: *Split goal for unique existential*

$$\vdash \forall\, \mathcal{P}_0,\ \mathcal{P}_1,\ x\ .$$
$$(\exists\, x\ .\ (\mathcal{P}_0\ x)\ \&\ (\mathcal{P}_1\ x))\ \&$$
$$(\exists!\ x\ .\ \mathcal{P}_0\ x) \implies$$
$$(\exists!\ x\ .\ (\mathcal{P}_0\ x)\ \&\ (\mathcal{P}_1\ x))$$

Lemma 8.18, which is based on the control specifications and says that all paths through the combinational representation lead to an exit (*termComb*, Definition 8.4 in Section 8.1.3). We then use Lemma 8.19 to transform the goal from "every input matches exactly one output" to "every input traverses the pipeline and exits exactly once." We capture the notion of traversing the pipeline and exiting exactly once in *termPipe* (Definition 8.7). Lemma 8.19 is derived from the definition of *termPipe* and the ordering specification. The proof continues in Figure 8.4 (hence the "(8.4)" before the turnstile in the goal). The body of the proof is concerned with showing that every transaction that enters the pipeline traverses it and exits exactly once.

In *termPipe*, the transaction that enters segment s_0 at time t_0 terminates if there is exactly one time t_n such that the transaction traverses from s_0 at t_0 to *Exit* at t_n and transfers from *Exit* to *Bot* at t_n. The definition of *termPipe* has as an antecedent that a transaction transfers into s_0 at t_0, because it is convenient to say that *termPipe* is true for all segments at all times. If we did not include the antecedent, we would have to qualify every use of *termPipe* with a test to check if a transaction transfers into the segment at the given time. If a transaction does not transfer into the given segment (s_0) at the given time (t_0), then there is no time t_n such that *travPipe* is true for a path from s_0 at t_0 to *Exit* at t_n. We require that there is exactly one exit time because part of our pipelining correctness criteria (*PipeliningOk*, Definition 7.8) is that there is a one-to-one correspondence between input and output transactions. We use *termPipe* in the proof that every input transaction produces exactly one output, which is described in Section 8.2.3.

The relation *termPipe* is defined in terms of *travPipe*, which is defined inductively. As shown

Lemma 8.17: *Transactions transfer to the correct next segment*

$$\vdash \forall s_0,\ s_1,\ t_0,\ t_1,\ T\ .$$
$$match\ s_0\ (t_0,\ t_1)\ \&$$
$$segComb\ s_0\ (transP\ s_0\ t_0)\ (T,\ s_1) \implies$$
$$xfr\ (s_0,\ s_1)\ t_1$$

Theorem 8.7: *Forward direction of one-to-one*
$\vdash \forall t_i \;.\; \mathit{XfrIn}\ t_i \implies \exists!\ t_o \;.\; \mathit{Match}\ I\ O\ t_i\ t_o$

Lemma 8.18: *Termination in combinational representation*
$\vdash \forall\ T_0 \;.\; \mathit{termComb}\ (\mathit{Entry},\ T_0)$

Lemma 8.19: *Termination in pipeline implementation*
$\vdash \forall t_0 \;.\; \mathit{termPipe}\ (\mathit{Entry},\ t_0) \implies$
$\quad (\mathit{XfrIn}\ t_0 \implies \exists!\ t_n \;.\; \mathit{Match}\ I\ O\ t_0\ t_n)$

Definition 8.7: *Termination for implementation of a pipeline*
$$\begin{array}{l} \mathit{termPipe}\ (s_0,\ t_0)\ \hat{=} \\ \quad \mathit{xfrIn}\ s_0\ t_0 \implies \\ \quad\quad \exists!\ t_n \;.\; \mathit{travPipe}\ (s_0,\ t_0)\ (\mathit{Exit},\ t_n)\ \ \&\ \ \mathit{xfr}\ (\mathit{Exit},\ \mathit{Bot})\ t_n \end{array}$$

$\vdash\ \forall t_0 \;.\; \mathit{XfrIn}\ t_0 \implies \exists!\ t_n \;.\; \mathit{Match}\ I\ O\ t_0\ t_n$

BY `Lemma 8.18 (*` $\forall T_0 \;.\; \mathit{termComb}\ (\mathit{Entry},\ T_0)$ `*)`

• $\forall\ T_0 \;.\; \mathit{termComb}\ (\mathit{Entry},\ T_0)$

$\vdash\ \forall t_0 \;.\; \mathit{XfrIn}\ t_0 \implies \exists!\ t_n \;.\; \mathit{Match}\ I\ O\ t_0\ t_n$

BY `Lemma 8.19 (*` $\mathit{termPipe} \implies (\mathit{XfrIn} \implies \mathit{Match})$ `*)`

• $\forall\ T_0 \;.\; \mathit{termComb}\ (\mathit{Entry},\ T_0)$

(8.4) $\vdash\ \forall\ t_0 \;.\; \mathit{termPipe}\ (\mathit{Entry},\ t_0)$

Figure 8.3: Theorem 8.7 — One-to-one forward; first step

- $\forall\ T_0\ .\ \mathit{termComb}\ (\mathit{Entry},\ T_0)$

$\vdash\ \forall\ t_0\ .\ \mathit{termPipe}\ (\mathit{Entry},\ t_0)$

BY `Lemma 8.8 with` $\mathcal{P}\ s_0\ T_0 = \forall\ t_0\ .\ \mathit{termPipe}\ (s_0,\ t_0)$
(* Induction lemma for *termComb* *)
`THEN Lemma 8.20` (* *termPipe* can be treated inductively *)

- $\forall\ T_0\ .$
  $\mathit{Entry}{=}\mathit{Exit}$ &
  $\exists\ T_n\ .\ \mathit{segComb}\ \mathit{Entry}\ T_0\ (\mathit{Exit},\ (T_n,\ \mathit{Bot}))$

$(-) \vdash\ \mathit{xfrIn}\ \mathit{Entry}\ t_0\ \Longrightarrow$
$\exists!\ t_n\ .$
$(\mathit{Entry} = \mathit{Exit})$ & $(\mathit{match}\ \mathit{Entry}\ (t_0,\ t_n))$ &
$(\mathit{xfr}\ (\mathit{Exit},\ \mathit{Bot})\ t_n$

- $\forall\ T_0\ .$
  $\exists\ s_x,\ T_x\ .$
  $(\mathit{segComb}\ \mathit{Entry}\ T_0\ (T_x,\ s_x))$ &
  $(\forall\ t_x\ .\ \mathit{termPipe}\ (s_x,\ t_x))$

$(8.5) \vdash\ \mathit{xfrIn}\ \mathit{Entry}\ t_0\ \Longrightarrow$
$\exists!\ s_x,\ t_x\ .$
$(\mathit{match}\ \mathit{Entry}\ (t_0,\ t_x))$ &
$(\mathit{xfr}\ (\mathit{Entry},\ s_x)\ t_x)$ &
$(\mathit{termPipe}\ (s_x,\ t_x))$

Figure 8.4: Theorem 8.7 — One-to-one forward; perform induction on *termComb*

in Lemma 8.20, we can treat *termPipe* as if it was defined inductively. Lemma 8.20 is proved by unfolding the definitions of *travPipe* and performing simple logical manipulations of the formulas.

Lemma 8.20: *Termination in a pipeline is inductive*

$$
\begin{array}{l}
\vdash \forall\, s_0,\ t_o\ .\\
\quad \mathit{termPipe}\ (s_0,\ t_0) \implies\\
\qquad (\mathit{xfrIn}\ s_0\ t_0 \implies\\
\qquad\quad \exists!\ t_n\ .\\
\qquad\qquad s_0 = \mathit{Exit}\ \&\\
\qquad\qquad \mathit{match}\ s_0\ (t_0,\ t_n)\ \&\\
\qquad\qquad \mathit{xfr}\ (\mathit{Exit},\ \mathit{Bot})\ t_n\\
\qquad)\ \vee\\
\qquad (\mathit{xfrIn}\ s_0\ t_0 \implies\\
\qquad\quad \exists s_x\ .\ \exists!\ t_x\ .\\
\qquad\qquad \mathit{match}\ s_0\ (t_0,\ t_x)\ \&\\
\qquad\qquad \mathit{xfr}\ (s_0,\ s_x)\ t_x\ \&\\
\qquad\qquad \mathit{termPipe}\ (s_x,\ t_x)\\
\qquad\quad)
\end{array}
$$

In Figure 8.4 we use Lemma 8.8 to perform induction on *termComb*. This effectively performs induction on the length of the path that the transaction follows through the pipeline. We apply Lemma 8.20 and choose the base case of *termPipe* in the base case of the proof and the inductive case of *termPipe* in the inductive case of the proof. In the base case the entry and exit segments of the path are the same, the transaction uses *Entry*, which then selects the bottom environment as the next segment. In the inductive case the transaction uses *Entry*, transfers to an intermediate segment, and traverses from it to *Exit*. The inductive hypothesis says if a transaction successfully transfers to the correct intermediate segment then the transaction will traverse the rest of the way through the pipeline.

The proofs of the base and inductive cases are similar. The base case is simpler, so we do not present it here. The inductive case is shown in Figure 8.5. The complete proof of Theorem 8.7 is included in Appendix D.6.

The inductive hypothesis says that for every input transaction T_0, there is an intermediate segment s_x and transaction T_x such that the combinational representation of *Entry* selects s_x as the next segment and all transactions that enter s_x are guaranteed to terminate. The goal says that if a transaction transfers into *Entry* at time t_0, there exists an intermediate time (t_x) such that the transaction uses *Entry* from t_0 to t_x, transfers from *Entry* to s_x at t_x and then terminates from s_x at t_x. By "terminates", we mean that the transaction traverses from s_x at t_x to *Exit* and then transfers from *Exit* to *Bot*.

From the antecedent in the conclusion of Figure 8.5, we know that a transaction transfers into *Entry* at t_0. We instantiate T_0 in the hypothesis with the input transaction to *Entry*

- $\forall\ T_0\ .\ \exists s_x,\ T_x\ .$
  $segComb\ Entry\ T_0\ (T_x,\ s_x)$ &
  $\forall\ t_x\ .\ termPipe\ (s_x,\ t_x)$

$\vdash$ $xfrIn\ Entry\ t_0 \implies$
$\exists!\ s_x,\ t_x\ .$
$(match\ Entry\ t_0\ t_x)$ & $(xfr\ (Entry,\ s_x)\ t_x)$ & $(termPipe\ (s_x,\ t_x)$

BY `Instantiate hyp with` $T_0{=}(transP\ Entry\ t_0)$

- $xfrIn\ Entry\ t_0$
- $segComb\ Entry\ (transP\ Entry\ t_0)\ (T_x,\ s_x)$
- $\forall\ t_x\ .\ termPipe\ (s_x,\ t_x)$

$\vdash$ $\exists!\ s_x,\ t_x\ .$
$(match\ Entry\ t_0\ t_x)$ & $(xfr\ (Entry,\ s_x)\ t_x)$ & $(termPipe\ (s_x,\ t_x))$

BY `Lemma 8.14`
(* Transfer into a segment implies *)
(* transfer to correct next segment *)

- $\exists!\ t_x\ .\ (match\ Entry\ t_0\ t_x)$ & $(xfr\ (Entry,\ s_x)\ t_x)$
- $\forall t_x\ .\ termPipe\ (s_x,\ t_x)$

(8.6) $\vdash$ $\exists!\ s_x,\ t_x\ .$
$(match\ Entry\ t_0\ t_x)$ & $(xfr\ (Entry,\ s_x)\ t_x)$ & $(termPipe\ (s_x,\ t_x))$

Figure 8.5: Theorem 8.7 — One-to-one forward; inductive case

Lemma 8.21: *Destination segment for transfers are unique*
$\vdash\ \forall\ s_0,\ t\ .\ (\exists\ s_1\ .\ (xfr\ (s_0,\ s_1)\ t)) \implies (\exists!\ s_1\ .\ xfr\ (s_0,\ s_1)\ t)$

- $\exists!\ t_x\ .\ (\mathit{match\ Entry}\ t_0\ t_x)\ \&\ (\mathit{xfr}\ (\mathit{Entry},\ s_x)\ t_x)$
- $\forall t_x\ .\ \mathit{termPipe}\ (s_x,\ t_x)$

$\vdash\ \exists!\ s_x,\ t_x\ .$
$(\mathit{match\ Entry}\ t_0\ t_x)\ \ \&\ \ (\mathit{xfr}\ (\mathit{Entry},\ s_x)\ t_x)\ \ \&\ \ (\mathit{termPipe}\ (s_x,\ t_x))$

BY `Lemma 8.16 (*` split goal for unique existential quantifier `*)`
`with` $\mathcal{P}_0\ (s,\ t) = (\mathit{match\ Entry}\ t_0\ t)\ \&\ (\mathit{xfr}\ (\mathit{Entry},\ s)\ t)$,
$\mathcal{P}_1\ (s,\ t) = \mathit{termPipe}\ (s,\ t)$

- $(\mathit{match\ Entry}\ t_0\ t_x)\ \&\ (\mathit{xfr}\ (\mathit{Entry},\ s_x)\ t_x)$
- $\forall t_x\ .\ \mathit{termPipe}\ (s_x,\ t_x)$

$\vdash\ \exists\ s_x,\ t_x\ .$
$(\mathit{match\ Entry}\ t_0\ t_x)\ \ \&\ \ (\mathit{xfr}\ (\mathit{Entry},\ s_x)\ t_x)\ \ \&$
$(\mathit{termPipe}\ (s_x,\ t_x))$

BY `Instantiate concl with` $s_x{=}s_x$ `,` $t_x{=}t_x$
`THEN Instantiate hyp with` $t_x{=}t_x$

- $(\mathit{match\ Entry}\ t_0\ t_x)\ \ \&\ \ (\mathit{xfr}\ (\mathit{Entry},\ s_x)\ t_x)$
- $\mathit{termPipe}\ (s_x,\ t_x)$

$\vdash\ (\mathit{match\ Entry}\ t_0\ t_x)\ \&\ (\mathit{xfr}\ (\mathit{Entry},\ s_x)\ t_x)\ \&$
$(\mathit{termPipe}\ (s_x,\ t_x))$
■

- $\exists!\ t_x\ .\ (\mathit{match\ Entry}\ t_0\ t_x)\ \&\ (\mathit{xfr}\ (\mathit{Entry},\ s_x)\ t_x)$

$\vdash\ \exists!\ s_x,\ t_x\ .\ (\mathit{match\ Entry}\ t_0\ t_x)\ \ \&\ \ (\mathit{xfr}\ (\mathit{Entry},\ s_x)\ t_x)$

BY `Lemma 8.21`
`(*` $(\exists\ s_x\ .\ \mathit{xfr}\ (\mathit{Entry},\ s_x)\ t_x)\ \ \Longrightarrow$ `*)`
`(*` $(\exists!\ s_x\ .\ \mathit{xfr}\ (\mathit{Entry},\ s_x)\ t_x)$ `*)`

- $\exists!\ s_x,\ t_x\ .\ (\mathit{match\ Entry}\ t_0\ t_x)\ \ \&\ \ (\mathit{xfr}\ (\mathit{Entry},\ s_x)\ t_x)$

$\vdash\ \exists!\ s_x,\ t_x\ .\ (\mathit{match\ Entry}\ t_0\ t_x)\ \ \&\ \ (\mathit{xfr}\ (\mathit{Entry},\ s_x)\ t_x)$
■

Figure 8.6: Theorem 8.7 — One-to-one forward; proving existence and uniqueness of path

---

at t_0. This gives us as a hypothesis that the combinational representation of *Entry* will select s_x as the next segment and that all transactions that enter s_x will terminate. We use Lemma 8.14 to prove that the transaction that transfers into *Entry* at t_0 will exit and transfer to s_x. At this point we know that there is exactly one time (t_x) such that the transaction that enters *Entry* at t_0 exits from *Entry* at t_x and transfers to s_x and that for s_x, *termPipe* is true for all times. To solve the goal, we need to prove that there is exactly one segment (s_x) and time (t_x) meeting all three requirements (*match*, *xfr*, and *termPipe*). That is, we must prove both existence and uniqueness for s_x and t_x.
As shown in Lemma 8.16 in Section 8.2.2, to prove uniqueness and existence for multiple conjuncts, it is sufficient to prove existence for all of the conjuncts together and uniqueness for one conjunct. From the hypotheses we know that there is a unique s_x and t_x for which the first two conjuncts in the conclusion are true. In Figure 8.6 we divide the goal into a existence proof for *match*, *xfr*, and *termPipe* and a uniqueness proof for *match* and *xfr*. We solve the existence subgoals by instantiating s_x and t_x in the conclusion and t_x in the hypothesis. In the uniqueness proof, we already know that there is a unique time at which the transaction transfers from *Entry* to s_x. We need to prove that there is only one segment to which the transaction could transfer. We solve the goal using Lemma 8.21 to say that each transfer out of a segment has a single destination. Note, Lemma 8.21 says that each transfer has a single destination segment. This does not preclude pipelines in which different transactions follow different paths. The lemma follows from the definition of *xfr* and the observation that each transaction selects exactly one segment as its next segment.

8.2.4 All Output Transactions Have Traversed the Pipeline

In this section we prove that every transaction that exits from the pipeline results from exactly one input transaction (Theorem 8.8). This is the second of the two directions in proving that there is a one-to-one mapping between input and output transactions. Much of the reasoning uses the same techniques as in the proof that every input transaction matches exactly one output (Section 8.2.3). For this reason, in this section, we include the principal lemmas and brief descriptions of the reasoning without showing the actual proof steps. To prove Theorem 8.8, we begin with several steps analogous to those done at the beginning of the proof that every input transaction produces exactly one output (Theorem 8.7). These steps allow us to prove Theorem 8.8 from 8.9. We use the ordering specification to convert the conclusion of Theorem 8.9 to that of Theorem 8.8. The the conclusion in Theorem 8.9 reasons about traversing the pipeline and the conclusion of Theorem 8.8 is written in terms of the matching relation for the pipeline. The ordering specification says that *Match* returns true for transactions that traverse the pipeline. The antecedent in Theorem 8.9 says that the theorem is only true for segments and times along paths that terminate. In the proof of Theorem 8.8, we instantiate s_n in 8.9 with *Exit*, which trivially satisfies the antecedent. In Section 8.2.4.1 we use the antecedent in Theorem 8.9 to prove that paths that terminate do not contain any combinational loops, that is, cyclic paths whose latency is zero clock

Theorem 8.8: *Each output transaction results from exactly one input*

$$\vdash \forall t_n \;.\; \mathit{XfrOut}\ t_n \implies \exists!\ t_0 \;.\; \mathit{Match}\ I\ O\ t_0\ t_n$$

Theorem 8.9: *All paths originate from top environment*

$$\begin{array}{l} \vdash \forall\ t_n,\ s_n\ . \\ \qquad (\mathit{termPipe}\ (s_n,\ t_n))\ \&\ (\mathit{xfrOut}\ s_n\ t_n) \implies \\ \qquad\quad \exists!\ t_0\ .\ (\mathit{xfr}\ (\mathit{Top},\ \mathit{Entry})\ t_0)\ \&\ (\mathit{travPipe}\ (\mathit{Entry},\ t_0)\ (s_n,\ t_n)) \end{array}$$

cycles. From this, we prove in Section 8.2.4.2 that all paths contain a previous point at which the transaction entered the pipeline or entered a segment at an earlier time. We combine these results together in Section 8.2.4.3 to prove Theorem 8.9.

8.2.4.1 Pipelines Do Not Have Combinational Loops

If a segment (s_0) and time (t_0) are part of a combinational loop, and there is a path from (s_0, t_0) to *Bot*, then for one of the segments (s_x) in the loop, we have contradictory values for the identity of its next segment at time t_0. Because the loop is combinational, no time transpires while a transaction traverses the loop. Thus, the next segment for s_x is the next segment in the loop (s_y). In order for a transaction to exit the loop, s_x must choose a segment not in the loop (s') as its next segment. Both s_y and s' are the selected next segment for s_x at time t_0. This contradicts the fact that each segment has exactly one next segment at a time, as shown in Lemma 8.21.

Lemma 8.22: *No combinational loops*

$$\begin{array}{l} \vdash \forall\ s,\ t_0,\ t_1\ . \\ \qquad (\mathit{termPipe}\ (s,\ t_0))\ \&\ (\mathit{travPipe}\ (s,\ t_0)\ (s,\ t_1)) \implies \\ \qquad\quad ((\mathit{match}\ s\ t_0\ t_1)\ \vee\ (t_0 < t_1)) \end{array}$$

We use the reasoning outlined above to prove Lemma 8.22, which says that if the path from (s, t_0) terminates, then any path that traverses from s to s must either contain only a single pass through s, or the entry time to s must be less than the exit time. We use *travPipe* for the path between (s, t_0) and (s, t_n), so there may be multiple segments in the path between the first entry to s at t_0 and the exit from s at t_1. The proof of Lemma 8.22 relies on the requirement in *CoreOk* that the latency through each segment is always non-negative.

Our qualification that we are only concerned with paths that lead to the exit is not a limitation. The segments in these paths are a superset of the segments that can be reached by transactions entering the pipeline, because in Section 8.2.3 we proved that every transaction that enters the pipeline eventually exits. So, in all reasonable pipelines, we in fact prove that each transaction within the pipeline (not just those that exit) originates from exactly one input transaction.

8.2.4.2 All Paths Decrement Time or Originate from Top Environment

In this section we prove that every transaction that transfers out of a segment (s_n) at time t_n has followed a finite length path where it either entered the pipeline from the environment or entered a segment at a time previous to t_n (Lemma 8.23). Not all paths must contain a point earlier in time, because we allow purely combinational segments and pipelines. Zero latency paths are valid as long as they originate from the top environment.

Lemma 8.23: *All paths come from top or have a point that is earlier in time*

$$
\begin{array}{l}
\vdash \forall\ s_n,\ t_n\ . \\
\qquad \mathit{termPipe}\ (s_n,\ t_n)\ \&\ \mathit{xfrOut}\ s_n\ t_n \implies \\
\qquad\quad \exists\ s_x,\ t_x\ .\ \mathit{travPipe}\ (s_x\ t_x)\ (s_n,\ t_n)\ \& \\
\qquad\qquad\qquad ((t_x < t_n)\ \vee\ (s_x = \mathit{Entry}\ \&\ \mathit{xfr}\ (\mathit{Top},\ \mathit{Entry})\ t_x))
\end{array}
$$

We prove Lemma 8.23 in two steps. First we prove that the maximum length of any combinational path is the number of segments in the pipeline (Lemma 8.24). We then prove that for any n, either there is a path back to the environment whose length is less than or equal to n or there is a segment n+1 steps previous in its path (Lemma 8.27).
We use *travPipeLength* (Definition 8.8) to determine the length of a path. It is identical to *travPipe* (Definition 7.20), except that we decrement a counter (*pl*, for path length) at each step. We use *travPipeLength* in Lemma 8.24 to prove that the longest path we can traverse through solely combinational segments is at most the number of segments in the pipeline (*numSegments*). If we were to traverse a longer combinational path, we would have to repeat a segment, which would contradict Lemma 8.22 (no combinational loops). Lemma 8.24 is proved using Lemma 8.22 and by performing induction on the number of segments in the pipeline.

Definition 8.8: *Traverse a path of a given length*

$$
\begin{array}{l}
\mathit{travPipeLength}\ (s_0,\ t_0)\ (s_n,\ t_n)\ \mathit{pl}\ \hat{=} \\
\qquad (s_0 = s_n)\ \&\ (\mathit{match}\ s_0\ (t_0,\ t_n))\ \&\ (\mathit{pl} = 1) \\
\qquad \vee \\
\qquad \exists !\ s_x,\ t_x\ . \\
\qquad\quad \mathit{match}\ s_0\ (t_0,\ t_x)\ \& \\
\qquad\quad \mathit{xfr}\ (s_0,\ s_x)\ t_x\ \& \\
\qquad\quad \mathit{travPipeLength}\ (s_x,\ t_x)\ (s_n,\ t_n)\ (\mathit{pl}{-}1)
\end{array}
$$

We now introduce *initPipeLength* (Definition 8.9), which determines the length of the path that a transaction has traversed from the top environment. We use *initPipeLength* in Lemmas 8.25 and 8.27 to say that if a transaction has not followed a path of length n or less from the top environment, then there is an intermediate segment n+1 steps previous in the path. More formally, there is a segment s_x and time t_x, such that the transaction

Lemma 8.24: *Maximum length of combinational paths is the number of segments*

$$\vdash \forall\ s_0,\ s_n,\ t,\ pl\ .\ termPipe\ (s_0,\ t) \implies$$
$$travPipeLength\ (s_0,\ t)\ (s_n,\ t)\ pl \implies (pl \leq numSegments)$$

Definition 8.9: *Initiation for implementation of a pipeline with path length*

$$initPipeLength\ (s_n,\ t_n)\ pl\ \hat{=}$$
$$xfrOut\ s_n\ t_n \implies$$
$$\exists!\ t_0\ .\ xfr\ (Top,\ Entry)\ t_0\ \ \&\ \ travPipeLength\ (Entry,\ t_0)\ (s_n,\ t_n)\ pl$$

traversed from s_x at t_x to s_n at t_n in n+1 steps. Lemma 8.25 is proved with induction on n and Lemma 8.26. In the base case (n=1) we unfold *initPipeLength* and apply Lemma 8.26 twice to produce an intermediate segment and time two steps away, which we then use to instantiation the conclusion. In the inductive case, the inductive hypothesis says that if there are no paths of length n or less back to *Top*, then there is a segment s_x that is n+1 steps away. There are no paths of length n+1 or less back to *Top*, so there are no paths of length n or less. We use the inductive hypothesis to produce a segment s_x and time t_x that are n+1 steps away and then apply Lemma 8.26 to produce the segment and time previous to s_x and t_x. This segment and time are n+2 steps away and so we have proved the lemma.

Lemma 8.25: *All paths not coming from top have an intermediate segment*

$$\vdash \forall\ n,\ s_n,\ t_n\ .$$
$$xfrOut\ s_n\ t_n\ \&$$
$$(\forall\ pl : \{1..n\}\ .\ \neg(initPipeLength\ (s_n,\ t_n)\ pl)\ ) \implies$$
$$\exists!\ s_x,\ t_x\ .\ travPipeLength\ (s_x,\ t_x)\ (s_n,\ t_n)\ (n{+}1)$$

Lemma 8.26 is similar to Lemma 8.14 which was used in the proof that every input transaction produces exactly one output (Theorem 8.7), except that, because we are following a path backwards, we do not need to worry about arbitration. We know that the transaction exited the segment (s_n) and *CoreOk* says that each core *satisfies* its combinational representation, which means that every output transaction from s_n results from exactly one input.

We now prove Lemma 8.23 from Lemma 8.27, which is derived from Lemma 8.25, and Lemma 8.24, which says that the maximum length of any zero-latency path is the number of segments in the pipeline. Lemma 8.23 says that every path starts from *Top* or contains a step that occurred earlier in time. This means that when a transaction exits from s_n at t_n, we can trace its path backwards and reach a point (s_x, t_x) where either t_x is earlier than t_n or s_x is *Entry* and the transaction transferred *Top* to *Entry* at t_x. We instantiate n in Lemma 8.27 with *numSegments*. This gives us two cases to consider, either there is a path whose length is at most *numSegments* and which starts at *Top*, or there is a path

Lemma 8.26: *Transfer out of Top or another segment*

$$\vdash \forall\ s_n,\ t_n\ .\ \mathit{xfrOut}\ s_n\ t_n \implies$$
$$s_n = \mathit{Top}\ \vee$$
$$(\exists!\ s_x,\ t_x\ .\ \mathit{match}\ s_n\ t_x\ t_n\ \ \&\ \ \mathit{xfr}\ (s_x,\ s_n)\ t_x)$$

Lemma 8.27: *All paths come from top, or have an intermediate segment*

$$\vdash \forall\ n,\ s_n,\ t_n\ .$$
$$\mathit{xfrOut}\ s_n\ t_n \implies$$
$$(\exists\ pl : \{1..n\}\ .\ (\mathit{initPipeLength}\ (s_n,\ t_n)\ pl)\ )\ \vee$$
$$\exists!\ s_x,\ t_x\ .\ \mathit{travPipeLength}\ (s_x,\ t_x)\ (s_n,\ t_n)\ (n{+}1)$$

whose length is at least *numSegments*+1. The first case is solved trivially, because the hypothesis says that there is a path back to *Top* whose length is at most *numSegments* and the left hand side of the disjunct in the goal says that there is a path that originates from *Top*. In the second case, we apply Lemma 8.24 to show that the point in the path that is *numSegments*+1 steps back $(s_x,\ t_x)$ must have occurred earlier in time $(t_x{<}t_n)$. This solves the left hand side of the disjunct in the conclusion.
The transformation from Lemma 8.25 to Lemma 8.27 requires negating a universal quantifier to convert it to an existential quantifier. In general, this conversion is not constructively true. It is valid in this case, because we are quantifying over a finite domain (the integers from 1 to n) and the predicate in the body of the quantifier is decidable.

8.2.4.3 All Paths Originate from Top Environment

Theorem 8.9 says that all transactions that exit the pipeline have paths that lead back to the top environment. The proof performs strong induction on the time (t_n) that the transaction exits the segment (s_n), and applies Lemma 8.23. From Lemma 8.23, we know that either there is a point in the path from an earlier time or the path began at *Top*. At time zero there is no earlier time, so the path began at *Top*. This is equivalent to the conclusion and so we complete the base case. In the inductive case, any transaction that exits a segment at time less than t_n is guaranteed to have originated from *Top*. Applying Lemma 8.23 leads us to a case split on whether the transaction's path leads to an intermediate segment at an earlier point in time or back to *Top*. In the first case we apply the inductive hypothesis and are done. The second case is analogous to the base case, in that the hypothesis says that the path began at *Top*, which is equivalent to the conclusion. This completes the proof of Theorem 8.9. This also completes the discussion of the proof of Theorem 8.8, which guarantees that every output transaction matches exactly one input.

8.3 Functionality

In this section we prove Theorem 8.10, which says that when a matching relation (*Match*) matches an input time (t_0) with an output time (t_n), the transaction that entered the pipeline at t_0 and the one that exited at t_n are correct with respect to the combinational representation.

Theorem 8.10: *Functionality for framework*

$$\vdash \forall t_0, t_n \ . \ Match\ I\ O\ t_0\ t_n \implies$$
$$Comb\ (getTransComb\ (I\ t_0))\ (getTransComb\ (O\ t_n))$$

We begin the proof in Figure 8.7 by using *CombOk* (Definition 8.10) to replace *Comb* with *travComb* (traversing between the combinational representations of the segments) and then using the ordering specification (Specification 8.5) to replace $Match\ I\ O\ t_0\ t_n$ with *travPipe* (traversing between segments in the implementation of the pipeline). The relations *travComb* and *travPipe* are defined in Sections 7.4 and 7.5.

We then use Lemmas 8.28 and 8.29 to replace entries in the input and output streams of the pipeline ($getTransComb\ (I\ t_0)$ and $getTransComb\ (O\ t_n)$) with the input and output transactions to the entry and exit segment of the pipeline ($transP\ Entry\ t_0$ and $transN\ Exit\ t_n$). Recall that we are working inside a context that includes the pipeline, input stream, output stream, and several predicates describing the pipeline (*e.g. CombOk*). The relation *CombOk* does not appear as an antecedent in Theorem 8.10, or as a hypothesis in Figure 8.7, but we are able to use it in the proof. Similarly, I and O are bound in the context and appear as free variables in Lemmas 8.28 and 8.29.

In Figure 8.8, we perform induction on *travPipe* using Lemma 8.7 and produce two subgoals. In the base case we know that the transaction uses *Entry* from t_0 to t_n and then transfers to *Bot*. We need to prove that the input transaction to *Entry* at t_0 ($transP\ Entry\ t_0$) and the output transaction from *Entry* at t_n ($transN\ Entry\ t_n$) are correct with respect to the combinational representation of *Entry*.

In the inductive case we have three hypotheses, the third of which is the inductive hypothesis. The first two hypotheses say that the transaction uses *Entry* from t_0 to some time (t_x) and then transfers to an intermediate segment (s_x) at t_x. From the inductive hypothesis we know that the input transaction to s_x at t_x ($transP\ s_x\ t_x$) and the output transaction from *Exit* at t_n ($transN\ Exit\ t_n$) are correct with respect to traversing from the combinational representations of the segments of s_x to *Exit*. The goal is to prove that there is an intermediate segment s_x and transaction T_x, such that the combinational representation of *Entry* produces T_x and selects s_x and then traversing the combinational representations from s_x to *Exit* produces the same output transaction as in the inductive hypothesis.

From Lemma 8.30, we know that when a transaction exits a segment its contents are correct and it requests the correct next segment. The correctness of the output data and selected next segment is determined by the combinational representation. Lemma 8.30 is proved from Lemmas 8.31 and 8.32, the control specifications, and the definition of *segComb*.

- $Match\ I\ O\ t_0\ t_n$

$\vdash$ $Comb\ (getTransComb\ (I\ t_0))\ (getTransComb\ (O\ t_n))$

BY `Unfold Definition 8.10 (*` $CombOk$ `*)`
`THEN Specification 8.5`

- $xfr\ (Top,\ Entry)\ t_0$
- $travPipe\ (Entry,\ t_0)\ (Exit,\ t_n)$
- $xfr\ (Exit,\ Bot)\ t_n$

$\vdash$ $travComb\ (Entry,\ (getTransComb\ (I\ t_0)))$
$(Exit,\ (getTransComb\ (O\ t_n)),\ Bot))$

BY `Lemma 8.28 (*` $(...(I\ t_0)) = (transP\ Entry\ t_0)$ `*)`
`THEN Lemma 8.29 (*` $(...(O\ t_n)) = (transN\ Exit\ t_n)$ `*)`

- $travPipe\ (Entry,\ t_0)\ (Exit,\ t_n)$
- $xfr\ (Exit,\ Bot)\ t_n$

(8.8) $\vdash$ $travComb\ (Entry,\ transP\ Entry\ t_0)\ (Exit,\ (transN\ Exit\ t_n,\ Bot))$

Figure 8.7: Theorem 8.10 — functionality proof; first step

Specification 8.5: *Ordering (Copy of Specification 7.18)*

$\forall I,\ t_0,\ O,\ t_n\ .$
$\quad Match\ I\ O\ t_0\ t_n \iff$
$\quad\quad xfr\ (Top,\ Entry)\ t_0\ \&$
$\quad\quad travPipe\ (Entry,\ t_0)\ (Exit,\ t_n)\ \&$
$\quad\quad xfr\ (Exit,\ Bot)\ t_n$

Definition 8.10: *Combinational representation is correct (Copy of Definition 7.15)*

$CombOk\ Pipe\ Comb \mathrel{\hat{=}}$
$\quad local\ open\ Pipe\ in$
$\quad\quad \forall\ T_i,\ T_o\ .\ Comb\ T_i\ T_o \iff travComb\ (Entry,\ T_i)\ (Exit,\ (T_o,\ Bot))$
$\quad end$

Lemma 8.28: *Transfer from Top to Entry*

$\vdash \forall\ t\ .\ (xfr\ (Top,\ Entry)\ t) \implies (transP\ Entry\ t) = (getTransComb\ (I\ t))$

Lemma 8.29: *Transfer from Exit to Bot*
$\vdash \forall t\ .\ \ (\mathit{xfr}\ (\mathit{Exit},\ \mathit{Bot})\ t) \implies (\mathit{transN}\ \mathit{Exit}\ t) = (\mathit{getTransComb}\ (O\ t))$

- $\mathit{travPipe}\ (\mathit{Entry},\ t_0)\ (\mathit{Exit},\ t_n)$
- $\mathit{xfr}\ (\mathit{Exit},\ \mathit{Bot})\ t_n$

$\vdash$ $\mathit{travComb}\ (\mathit{Entry},\ \mathit{transP}\ \mathit{Entry}\ t_0)\ (\mathit{Exit},\ (\mathit{transN}\ \mathit{Exit}\ t_n,\ \mathit{Bot}))$

BY `Lemma 8.7 (*` Induction on *travPipe* `*)`
`with` $\mathcal{P}\ (s_0,\ t_0)\ (s_n,\ t_n) =$
$\mathit{travComb}\ (s_0,\ \mathit{transP}\ s_0\ t_0)\ (s_n,\ (\mathit{transN}\ s_n\ t_n,\ \mathit{Bot}))$
`THEN Unfold Definition 7.18 (*` *travComb* `*)`

- $\mathit{match}\ \mathit{Entry}\ t_0\ t_n$
- $\mathit{xfr}\ (\mathit{Exit},\ \mathit{Bot})\ t_n$

$\vdash$ $\mathit{segComb}\ \mathit{Entry}\ (\mathit{transP}\ \mathit{Entry}\ t_0)$
$(\mathit{transN}\ \mathit{Entry}\ t_n,\ \mathit{Bot})$

BY `Lemma 8.30 (*` $\mathit{match} \implies \mathit{segComb}$ `*)`

- $\mathit{segComb}\ \mathit{Entry}\ (\mathit{transP}\ \mathit{Entry}\ t_0)$
$(\mathit{transN}\ \mathit{Entry}\ t_n,\ \mathit{Bot}))$

$\vdash$ $\mathit{segComb}\ \mathit{Entry}\ (\mathit{transP}\ \mathit{Entry}\ t_0)$
$(\mathit{transN}\ \mathit{Entry}\ t_n,\ \mathit{Bot}))$

■

- $\mathit{match}\ \mathit{Entry}\ t_0\ t_x$
- $\mathit{xfr}\ (\mathit{Entry},\ s_x)\ t_x$
- $\mathit{travComb}\ (s_x,\ (\mathit{transP}\ s_x\ t_x))$
$(\mathit{Exit},\ (\mathit{transN}\ \mathit{Exit}\ t_n,\ \mathit{Bot}))$

(8.9) $\vdash$ $\exists s_x,\ T_x\ .$
$\mathit{segComb}\ \mathit{Entry}\ (\mathit{transP}\ \mathit{Entry}\ t_0,\ (T_x,\ s_x))\ \&$
$\mathit{travComb}\ (s_x,\ T_x)\ (\mathit{Exit},\ (\mathit{transN}\ \mathit{Exit}\ t_n,\ \mathit{Bot}))$

Figure 8.8: Theorem 8.10 — Functionality proof; induction on *travPipe*

Lemma 8.30: *Matching input/output transactions from segments are correct*
$\vdash \forall s,\ s',\ t_i,\ t_o\ .$
$\quad (match\ s\ t_i\ t_o)\ \&\ (xfr\ (s,\ s')\ t_o) \implies$
$\qquad segComb\ s\ (transP\ s\ t_i)\ ((transN\ s\ t_o),\ s')$

Lemma 8.31: *Match implies combinational representation of core*
$\vdash \forall s,\ t_i,\ t_o\ .$
$\quad match\ s\ t_i\ t_o \implies$
$\qquad comb\ s\ (cInstP\ s\ t_i,\ dataP\ s\ t_i)\ (dataN\ s\ t_o)$

We use Lemma 8.30 in the base case of Figure 8.8 and the first subgoal of Figure 8.9. In Figure 8.8, we use Lemma 8.30 to prove that when the transaction exits *Entry* the output transaction and selected next segment are correct with respect to the combinational representation. This completes the proof of the base case.
For the inductive case (Figure 8.9), we instantiate s_x and T_x in the conclusion with s_x and the output transaction from *Entry* at t_x. This gives us two subgoals. The first is to prove that the input transaction to *Entry* at t_0 and the output transaction at t_x are correct with respect to the combinational representation of *Entry*. The second goal is to prove that the output transaction from *Entry* at t_x produces the correct output transaction from *Exit* at t_n. The first goal is solved by applying the functionality lemma for segments (Lemma 8.30) to *Entry*.
The first hypothesis in the second subgoal is identical to the conclusion, except that the input transactions to s_x differ. In the hypothesis we have *transP* s_x t_x and in the goal we have *transN Entry* t_x. The second hypothesis says that a transfer from *Entry* to s_x occurs at t_x, so we use Lemma 8.33, which says that if a transfer occurs between segments then the input to the second segment is equal to the output of the first segment. Performing this substitution completes the proof of Theorem 8.10.

8.4 Conclusion

In this chapter we proved the principal theorem in the framework, which says that a pipeline that meets the specifications of the four framework parameters is a valid segment core. This is a stronger statement than that the pipeline *satisfies* its combinational representation,

Lemma 8.32: *Information field is preserved when traversing a segment*
$\vdash \forall s,\ t_i,\ t_o\ .$
$\quad (match\ s\ t_i\ t_o) \implies$
$\qquad ((infoN\ s\ t_o) = (infoP\ s\ t_i))\ \&\ ((zInstP\ s\ t_o) = (instP\ s\ t_i))$

- $match\ Entry\ t_0\ t_x$
- $xfr\ (Entry,\ s_x)\ t_x$
- $travComb\ (s_x,\ transP\ s_x\ t_x)\ (Exit,\ (transN\ Exit\ t_n,\ Bot))$

$\vdash\ \exists s_x,\ T_x\ .$
$segComb\ Entry\ (transP\ Entry\ t_0,\ (s_x,\ T_x))$ &
$travComb\ (s_x,\ T_x)\ (transN\ Exit\ t_x,\ Exit)$

BY `Instantiate concl with` $s_x{=}s_x$, $T_x{=}transN\ Entry\ t_x$

- $match\ Entry\ t_0\ t_x$
- $xfr\ (Entry,\ s_x)\ t_x$

$\vdash\ segComb\ Entry\ (transP\ Entry\ t_0)\ (transN\ Entry\ t_x,\ s_x))$

BY `Lemma 8.30 (*` $match \implies segComb$ `*)`

- $segComb\ Entry\ (transP\ Entry\ t_0)\ (transN\ Entry\ t_x,\ s_x)$

$\vdash\ segComb\ Entry\ (transP\ Entry\ t_0)\ (transN\ Entry\ t_x,\ s_x)$

■

- $travComb\ (s_x,\ transP\ s_x\ t_x)\ (Exit,\ (transN\ Exit\ t_n,\ Bot)$
- $xfr\ (Entry,\ s_x)\ t_x$

$\vdash\ travComb\ (s_x,\ transN\ Entry\ t_x)$
$(Exit,\ (transN\ Exit\ t_n,\ Bot))$

BY `Lemma 8.33`
`(*` $xfr\ (Entry,\ s_x)\ t_x \implies$ `*)`
`(*` $(transP\ s_x\ t_x\ =\ transN\ Entry\ t_x)$ `*)`

- $travComb\ (s_x,\ transP\ s_x\ t_x)\ (Exit,\ (transN\ Exit\ t_o,\ Bot))$

$\vdash\ travComb\ (s_x,\ transP\ s_x\ t_x)\ (Exit,\ (transN\ Exit\ t_o,\ Bot))$

■

Figure 8.9: Theorem 8.10 — Functionality theorem; inductive case

---

Lemma 8.33: *Transfer*
$\vdash\ xfr\ (s_1,\ s_2)\ t \implies (transP\ s_2\ t) = (transN\ s_1\ t)$

which is used in the proof that a pipeline *satisfies* a functional specification. Knowing that the pipeline is a valid core allows us to use the pipeline as the core of a segment in a hierarchical pipeline. This theorem greatly simplifies the verification of a pipeline and allows us to reuse proof results as we build up complex hierarchical pipelines.

We divided the proof of this theorem into eight subproofs. Four of these were very simple. The four substantial theorems were to prove that every sustained request to a segment in the pipeline is accepted, every input transaction to the pipeline produces exactly one output transaction, every output results from exactly one input, and every output transaction is correct with respect to the combinational representation. The three latter theorems were proved by performing induction on how a transaction traverses a path through an implementation or combinational representation.

Chapter 9

Conclusion

9.1 Summary of Work

The work presented here is aimed at developing general techniques for applying formal methods to the specification, design, and verification of pipelined circuits. We began with a formal definition of correctness for pipelines that guarantees that transactions are able to traverse through a pipeline and will exit with the correct data. This definition allowed us to separate the pipelining and functionality aspects 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. The definition of pipelining correctness is the same for every pipeline. We captured this commonality and 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. Using the specifications for the four framework parameters simplifies the process of verifying a pipeline, because the proof is broken up into a number of independent tasks.

To help guide the pipelining verification we identified a variety of schemes used in the pipelines of state-of-the-art commercial microprocessors to instantiate the parameters. We also provide decision trees to help determine which instantiations are valid for a given pipeline. As design styles evolve, the framework can be extended with new choices in the decision trees and additional standard implementations.

The framework defines virtual functional units whose behavior the pipeline stages implement. The virtual function units prescribe required behavior; they do not constrain how the behavior is implemented. Simple pipelines often implement the virtual functional units with very little hardware, while more complex pipelines require additional hardware.

Having introduced the framework parameters, virtual functional units, and decision trees, we worked through the specification, design, and informal verification of a simple pipeline with a structural hazard. This illustrated our techniques, including optimizing the hardware for virtual functional units and separating the pipelining and functionality aspects of verification. We demonstrated the use of the framework with complex pipelines by outlining the specification and verification of the instruction and floating-point pipelines of a fictitious microprocessor, the Adirondack. We based the Adirondack primarily on the Digital Alpha 21064. We showed how the framework can be used to reduce the verification of the datapath of a super-scalar pipeline to that of a single-issue pipeline. The description of the floating-point pipeline demonstrated the verification of a relatively complex pipeline with out-of-order termination and paths that looped through stages multiple times.

The formal foundation of the framework began with a detailed analysis of our definition of correctness. This definition guarantees that every transaction that wants to enter the pipeline is able to do so, there is a one-to-one correspondence between input and output transactions, and every pair of matching input/output transactions satisfies a functional specification. We then introduced the theorems used to separate the pipelining and functionality aspects of verification and to decompose the pipelining proof into three tasks. These tasks are to prove that the core of each segment is correct, the constraints each segment places on its environment are met, and the combinational representation of the pipeline implies the specification.

Proving that the core of each segment is correct is done by using the framework to verify hierarchical pipelines. Using the framework to verify a pipeline is sufficient to prove that the pipeline can be used as the core of a segment. In all pipelines that we have found, proving that the constraints of the segments are met (the second of the three tasks) is very straightforward. The third task is to prove that the combinational representation of the pipeline implies the specification. This proof does not reason about any pipelining concerns. It is purely combinational and can be handled by conventional hardware verification techniques.

The proof of the theorem used to separate the pipelining and functionality aspects of verification is very easy, but the proof that the framework specifications are sufficient to guarantee that the pipeline can be used as the core of a larger segment was quite involved. The proof relies on three main theorems about pipelines that conform to the framework specifications: every input transaction is matched with exactly one output, every output transaction matches exactly one input, and every pair of matching input/output transactions is correct with respect to the combinational representation.

9.2 Future Work

The framework represents our initial steps in creating a formal model of pipelines that will act as the foundation for a suite of verified tools for pipeline design and verification. These tools will allow engineers to explore design tradeoffs and build pipelines that exploit complex features to improve performance while increasing confidence in the correctness of the final product and reducing the resources required for design and verification. Towards that end we now describe four aspects of our proposed future work: mechanizing the verification of the framework, increasing the ease of verifying a pipeline, building CAD tools based on the framework, and incorporating control and data hazards into the framework.

9.2.1 Mechanizing the Verification of the Framework

The verification of the framework was carried out on paper, but in a style that is compatible with interactive, tactic-based proof development systems using a strongly-typed higher-order logic. We are confident that it will be straightforward to translate the proofs done on paper into a proof development system. The primary question is, given this opportunity to rework the proofs, how can we improve them? We believe that the overall style of reasoning and the technique of induction over paths are the right approach. There is room for improvement though, in some of the organizational aspects.

The hierarchy of theorems in the verification of the framework can be viewed as a tree, with the specifications of the framework parameters at the bottom and the principal theorems at the top. In the current proofs, there are no obvious lines that divide the tree into layers of independent proofs. This means that to fully understand the proofs, one must be familiar with all of the reasoning from the protocol specifications for connecting signals between segments to the induction lemmas used in the top level theorems.

We believe that we can find at least two well defined layers, where only a small number of theorems in the top layer depend on theorems in the lower layer. These theorems that span the division between the layers are really intermediate level specifications. These intermediate specifications will be very similar to the seven major specifications described in Section 7.6 and repeated here as Table 9.1.

By structuring the proofs with this clean division, readers can concentrate on the higher level proofs and be burdened with fewer details. The proofs that the low level specifications are sufficient to guarantee the intermediate specifications will be very routine. The interesting pipelining issues will all appear in the higher level proofs.

9.2.2 Verifying Pipelines

The specifications of the framework parameters were chosen so that automated techniques, such as symbolic simulation, model checking, or automated tactics can verify almost all of the specifications for large classes of pipelines. Therefore, our immediate plans are to demonstrate this capability with a number of example pipelines.

A longer range goal is to leverage off a mechanization verification of the framework and verify both the functionality and pipelining aspects of correctness within a single system or by connecting multiple systems together. The HOL proof system has been connected to Seger's symbolic simulator Voss [JS93] and to the computer algebra program Maple [HT93]. Once the verification of the framework has been mechanized, we will verify the correctness of the commonly used instantiations of the framework parameters. This will introduce a layer of specifications below that of the general framework specifications. It will be easier to verify a pipeline against the specifications of a verified instantiation than against the general specifications, because the reasoning that is common to pipelines using an instantiation will be captured in the proofs of the verified instantiations. Taken together, the variety of specifications and verification tools will offer many different methods for verifying pipelines. These methods will be able to be mixed and matched for the most

Table 9.1: Major specifications of framework parameters

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 efficient verification of each part of a pipeline.

9.2.3 CAD Tools

We have discussed using the framework to verify individual pipelines. The framework can also be used to verify CAD tools for designing pipelines. By verifying the tool rather than the pipeline that the tool produces, we will enable engineers unfamiliar with formal methods to gain the increased assurance that formal verification offers. We have found an algorithm for compiling the pipelining and control circuitry for pipelines using a static arbitration scheme [AL93a, AL93b]. This algorithm takes as input a textual description of the paths that transactions are to take through the pipeline and outputs register-transfer level descriptions of the pipelining and control circuitry. Davidson [Dav71] and Kogge [Kog81] have derived and verified algorithms for scheduling transactions in pipelines. They also describe techniques needed to build hardware implementing these schedules. These techniques can be automated and verified, thus offering users a choice of how to implement pipelining circuitry. As we increase the number of choices of implementations and verification methods available, we will see the realization of our ambition of a suite of coordinated design and verification tools.

9.2.4 Control and Data Hazards

Structural hazards can lead to errors with individual transactions, such as non-termination. Control and data hazards can lead to errors between transactions (*e.g.* reading incorrect data from a register file). To expand the framework to reason about control and data hazards, we need to develop a formal semantics for executing streams of interdependent transactions. These issues are dealt with in compilers with instruction scheduling [PBJ+91, PF94, HMC94] and so we expect to draw on work done in this area.

Based on our preliminary studies, we believe that control and data hazards can be added to the framework by building on the existing work with structural hazards. Control hazards affect whether a transaction that was issued to the pipeline is supposed to exit. Currently, all transactions that are issued must exit. With control hazards, a transaction should exit if and only if an earlier transaction that exits says the current transaction should exit. For example a non-control transaction says that the subsequent transaction in the input stream should exit and a conditional branch selects either the subsequent transaction or the transaction at the branch target.

Data hazards affect the transfer of data between transactions. The use of features such as bypass paths and register renaming means that we can not rely on data values appearing in specific locations before being used. Instead, we require that there is some method by which the data produced by a transaction is consumed by the dependent transactions. This will likely require tagging pieces of data (as we now do with transactions in some ordering schemes) and then performing fork and join operations to describe the production and consumption of data.

Throughout the development of the framework we sought out general specifications and then specialized them to capture commonly used solutions. This practice enabled us to define a library of instantiations to the framework parameters that is sufficient to characterize pipelines from state-of-the-part superscalar microprocessors. In the future, this generality will allow us to build and verify CAD tools based on the framework, incorporate control and data hazards, and expand our library of instantiations for the framework parameters to include microprocessors not yet even on the drawing boards.

Bibliography

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

[AAD+93] Tom Aspre, Gregory Averill, Eric DeLano, Russ Mason, Bill Weiner, and Jeff Yetter. Performance features of the PA7100 microprocessor. *IEEE Micro*, pages 22–35, June 1993.

[AL93a] Mark D. Aagaard and Miriam E. Leeser. An algorithm for compiling control circuitry for pipelines with data-dependent control. Technical Report TR EE-CEG-93-3, School of Electrical Engineering, Cornell University, March 1993.

[AL93b] Mark D. Aagaard and Miriam E. Leeser. A framework for specifying and designing pipelines. In *International Conference on Computer Design*, pages 548–551. IEEE Computer Society Press, Washington D.C., October 1993.

[B+93] Michael C. Becker et al. The PowerPC 601 microprocessor. *IEEE Micro*, pages 54–68, October 1993.

[Bar77] Jon Barwise. *Handbook of Mathematical Logic*. Elsevier Science Publishers, B.V.; Amsterdam, 1977.

[BBC+89] Bradley J. Benschneider, William J. Bowhill, Elizabeth M. Cooper, Moshe N. Gavrelov, Paul E. Gronowski, Vijay K. Maheshwari, Victor Peng, Jeffrey D. Pickholtz, and Sridhar Samudrala. A pipelined 50-MHz CMOS 64-bit floating-point arithmetic processor. *IEEE Journal of Solid-State Circuits*, 24(5):1317–1323, October 1989.

[BCF92] Simon Bainbridge, Albert Camilerri, and Roger Fleming. Theorem proving as an industrial tool for system level design. In V. Stravidou, T. F. Melham, and R. T. Boute, editors, *Theorem Provers in Circuit Design*, pages 253–274. Elsevier Science Publishers, B.V.; Amsterdam, 1992.

[BCJ90] Randal E. Bryant and Seger Carl-Johan. Formal verification of digital circuits using symbolic ternary system models. In Robert Kurshan and E. M. Clarke, editors, *Workshop on Computer-Aided Verification*, June 1990.

[BD94a] Vishal Bhagwati and Srinivas Devadas. Automatic verification of pipelined microprocessors. In *ACM/IEEE Design Automation Conference*, July 1994.

[BD94b] Jerry R. Burch and David L Dill. Automatic verification of pipelined microprocessor control. In *Workshop on Computer-Aided Verification*, July 1994.

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

[BM88] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, 1988. Volume 23 of Perspectives in Computing.

[BT89] Alexandre Bronstein and Carolyn L. Talcot. Formal verification of synchronous circuits based on string functional semantics: The seven Paillet circuits in boyer-moore. In *Workshop on Automatic Verifiation Methods for Finite State Systems*, 1989.

[BT90] Alexandre Bronstein and Carolyn L. Talcott. Formal verification of pipelines. In L. J. M. Claesen, editor, *Formal VLSI Specification and Synthesis*, pages 349–366. Elsevier Science Publishers, B.V.; Amsterdam, 1990.

[C+86] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall; Englewood Cliffs, NJ, 1986.

[CES83] Edward M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1983.

[CM92] Juanito Camilleri and Tom Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, University of Cambridge, Computer Laboratory, Aug 1992.

[DA92] Keith Deifendorff and Michael Allen. Organization of the Motorola 88110 superscalar RISC microprocessor. *IEEE Micro*, 12:40–63, April 1992.

[Dav71] E.S. 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.

[DSTP75] E.S. Davidson, L.E. Shar, A.T. Thomas, and J.H. Patel. Effective control of pipelined computers. In *Spring Compcon*, pages 181–184, 1975.

[FWC93] David A. Fura, Phillip J. Windley, and Gerald C. Cohen. Towards the formal verification of the requirements and design of a processor interface unit. Technical Report NASA Contractor Report 4522, Langley Research Center, NASA, 1993.

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

[HT93] John Harrison and Laurent Thery. Extending the HOL theorem prover with a computer algebra system to reason about the reals. In Carl Seger and Jeff Joyce, editors, *Higher Order Logic Theorem Proving and Its Applications*, pages 174–184. Springer Verlag; New York, August 1993.

[JS93] Jeff Joyce and Carl-Johann Seger. Linking BDD based symbolic evaluation to interactive theorem proving. In *ACM/IEEE Design Automation Conference*, June 1993.

[Kan88] Gerry Kane. *Mips Risc Architecture*. Prentice Hall; Englewood Cliffs, NJ, 1988.

[KM89] R.P. Kurshan and K McMillan. Structural induction theorem for processes. In *ACM Symposium on Principles of Distributed Computing*, pages 239–247, August 1989.

[Kog81] Peter Kogge. *The Architecture of Pipelined Computers*. McGraw–Hill Book Company, 1981.

[LB90] Wayne Luk and Geoffrey Brown. A systolic LRU processor and its top-down development. *Science of Computer Programming*, 15:217–233, 1990.

[MBB+86] T.K III. Miller, Bharat L. Bhuva, Russell L. Barnes, Jyy-Chiang Duh, Hsing-Bang Lin, and David E. Ven den Bout. The HECTOR microprocessor. In *International Conference on Computer Design*, pages 406–411, 1986.

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

[MG93] Thomas F. Melham and Michael J. C Gordon. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, New York, 1993.

[Mil83] Robin Milner. Calcui for synchrony and asynchrony. *Journal of Theoretical Computer Science*, 25:267–310, 1983.

[Mil89] Veljko Milutinovic. *High Level Language Computer Architecture*. Computer Science Press Inc., 1989.

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

[Mos74] Yiannis N. Moschoakis. *Elementary Induction on Abstract Structurs*. Elsevier Science Publishers, B.V.; Amsterdam, 1974.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press; Cambridge, MA, 1990.

[MWV92] Sunil Mirapuri, Michael Woodacre, and Nader Vasseghi. The Mips R4000 processor. *IEEE Micro*, pages 10–22, April 1992.

[PBJ+91] Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudgill, and Paul Stodghill. Dependence flow graphs. An algebraic approach to program dependencies. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, page 67, 1991.

[PF94] Todd Proebsting and Chris Fraser. Detecint pipeline structural hazards quickly. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1994.

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

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

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

[She86] Mary Sheeran. Retiming and slowdown in Ruby. In G. Milner, editor, *The Fusion of Hardware Design and Verification.* Elsevier Science Publishers, B.V.; Amsterdam, 1986.

[SOR93] N. Shankar, S. Owre, and J.M. Rushby. The PVS proof checker: A reference manual (draft). Technical report, SRI International, 1993.

[Sto90] Harold Stone. *High Performance Computer Architectures.* Addison-Wesley, 1990.

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

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

[VAAD93] Filip Van Aelten, Jonathon Allen, and Srinivas Devadas. Verification of relations between synchonous machines. *IEEE Transactions on Computer Aided Design*, 12(12):1947–1959, December 1993.

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

[Win91] Phillip J. Windley. The practical verification of microprocessor designs. In *IEEE COMPCON: Computer Society International Computer Conference*, pages 462–467. IEEE Computer Society Press, Washington D.C., February 1991.

Appendix A

Logical Notation

We use a strongly typed, higher-order logic and sequent style proofs. This is compatible with mechanical proof systems such as Nuprl [C$^+$86], HOL [MG93], and PVS [SOR93]. Our logical operators and syntax are shown in Table A.1; they are based closely upon those used in Nuprl and SML [MTH90]

Table A.1: Logical operators and symbols

&	Logical and
$\vee$	Logical or
$\Longrightarrow$	Implication
$\Longleftrightarrow$	Logical equivalence
$\forall\ x : T\ .\ \mathcal{P}$	For all x of type T, $\mathcal{P}$ is true
$\exists\ x : T\ .\ \mathcal{P}$	There exists an x of type T, such that P is true
$\exists!\ x : T\ .\ \mathcal{P}$	There exists exactly one x of type T, such that P is true
$\{i..j\}$	The integers between i and j, inclusive
$\lambda x\ .\ exp$	Lambda abstraction
$a \hat{=} exp$	Definition. The variable a is defined to be *exp*.
(x,y)	The pair of x and y
local–in–end	Local declarations
open recd	Add the internal declarations of *recd* to the current scope
if–then–else	Conditional choice

In our proof sequents, hypotheses are displayed as a bullet-delimited list above the goal, which is delimited by "$\vdash$." A list of hypotheses together with a goal comprise a sequent. Sequents are connected by proof rules, or tactics, (delimited by "**BY**"), which justify the transformation between the sequents. When a proof is completed we use the symbol "$\blacksquare$." For example, Figure A.1 shows the use of Modus Ponens in a simple-proof.
When the application of a tactic produces more than one subgoal, then we indent the

- A
- $A \implies B$

$\vdash B$

BY `ModusPonens`

- B

$\vdash B$

$\blacksquare$

Figure A.1: Use of Modus Ponens in a simple proof

- $A \implies B$

$\vdash B$

BY `ModusPonens`

$\vdash A$

- B

$\vdash B$

$\blacksquare$

Figure A.2: Use of Modus Ponens in a simple proof, producing two subgoals

subgoals as shown in Figure A.2. If an indented subgoal can be solved in one step, then we show the tactic call after the goal. If the proof of a subgoal is shown in a subsequent figure, then the number of that figure appears to the left of the turnstile for the goal. To make the proofs easier to follow, we often include comments, which are delimited by "`(*`" and "`*)`." Many of the comments show a sketch of the lemma or theorem that was applied. When we use a lemma or theorem, we show the instantiations of terms for the universally quantified variables if the instantiations are not obvious. The complete list of tactics used in the proofs presented in the thesis appears in Table A.2.

Table A.2: Tactics

`Unfold` *name*	Unfold the definition of *name*
`Theorem` *n*	Apply theorem number *n*
`Theorem` *n* `with` *var*`=`*term*	Apply theorem *n* and instantiate universally quantified *var* in theorem with *term*
`Lemma` *n*	Apply lemma number *n*
`Lemma` *n* `with` *var*`=`*term*	Apply lemma and instantiate universally quantified *var* in lemma with *term*
`Instantiate concl with` *var*`=`*term*	Instantiate the existentially quantified variable *var* in the conclusion with *term*
`Instantiate hyp with` *var*`=`*term*	Instantiate the universally quantified variable *var* in a hypothesis with *term*

Appendix B

Formal Verification of the DiffAdd Pipeline

This appendix contains several parts of the verification of the *diffAdd* pipeline, which was described in Chapter 5. Appendix B.1 shows the calculation and simplification of the combinational representation and Appendix B.2 contains several examples of traversing the implementation.

B.1 Calculate and Simplify Combinational Representation of *diffAdd*

In this section we calculate the combinational representation of the *diffAdd* pipeline. From the definition of *travComb* (Definition B.1), we know that we traverse from the entry segment to the exit segment. In *diffAdd*, the entry segment is *sub* and the exit segment is *add*, thus we begin with Equation B.1.

Definition B.1: *Traverse through a combinational representation (Copy of Definition 7.18)*

$$
\begin{aligned}
&\mathit{travComb}\ (s_0,\ T_0)\ (s_n,\ (T_n,\ s')) \ \hat{=}\\
&\quad (s_0 = s_n)\ \&\ (\mathit{segComb}\ s_0\ T_0\ (T_n,\ s'))\\
&\quad \vee\\
&\quad \exists s_x,\ T_x\ .\\
&\qquad \mathit{segComb}\ s_0\ T_0\ (T_x,\ s_x)\ \&\\
&\qquad \mathit{travComb}\ (s_x,\ T_x)\ (s_n,\ (T_n,\ s'))
\end{aligned}
$$

$$\mathit{travComb}\ (\mathit{sub},\ (I_1,\ I_2,\ B))\ (\mathit{add},\ (T_n,\ \mathit{Bot})) \tag{B.1}$$

We unfold *travComb* using the inductive case to produce Equation B.2.

$$\begin{aligned}
&\exists\ s_x,\ T_x.\\
&\ \ \mathit{segComb}\ \mathit{sub}\ (I_1,\ I_2,\ B)\ (T_x,\ s_x)\ \&\\
&\ \ \mathit{travComb}\ (s_x,\ T_x)\ (\mathit{add},\ (T_n,\ \mathit{Bot})
\end{aligned} \tag{B.2}$$

The combinational representation of *sub* is shown in Equation B.3.

$$\begin{aligned}
&\mathit{segComb}\ \mathit{sub}\ (I_1,\ I_2,\ B)\ (T_x,\ s_x) \hat{=}\\
&\ \ \mathit{comb}\ \mathit{sub}\ (I1,\ I_2,\ B)\ T_x,\ \&\\
&\ \ \mathit{if}\ I_1 \geq I_2\ \mathit{then}\ s_x = \mathit{add}\ \mathit{else}\ s_x = \mathit{neg}
\end{aligned} \tag{B.3}$$

The combinational representation of *sub* contains the combinational representation of the core (*comb sub*) and the specifications of the control circuitry. The control circuitry selects the next segment (*neg* or *add*) depending on if $I_1 \geq I_2$. Substituting Equation B.3 into Equation B.2 results in Equation B.4.

$$\begin{aligned}
&\exists\ s_x,\ T_x.\\
&\ \ \mathit{comb}\ \mathit{sub}\ (I_1,\ I_2,\ B)\ T_x\ \&\\
&\ \ (\mathit{if}\ I_1 \geq I_2\ \mathit{then}\ s_x = \mathit{add}\ \mathit{else}\ s_x = \mathit{neg})\ \&\\
&\ \ \mathit{travComb}\ (s_x,\ T_x)\ (\mathit{add},\ (T_n,\ \mathit{Bot}))
\end{aligned} \tag{B.4}$$

We simplify Equation B.4 by moving *travComb* into the *then* and *else* branches of the *if–then–else* and substituting $s_x = \mathit{add}$ and $s_x = \mathit{neg}$ (Equation B.5).

$$\begin{aligned}
&\exists\ T_x.\\
&\ \ \mathit{comb}\ \mathit{sub}\ (I_1,\ I_2,\ B)\ T_x\ \&\\
&\ \ \mathit{if}\ I_1 \geq I_2\\
&\ \ \mathit{then}\ \mathit{travComb}\ (\mathit{add},\ T_x)\ (\mathit{add},\ (T_n,\ \mathit{Bot}))\\
&\ \ \mathit{else}\ \mathit{travComb}\ (\mathit{neg},\ T_x)\ (\mathit{add},\ (T_n,\ \mathit{Bot}))
\end{aligned} \tag{B.5}$$

The *travComb* in the *then* side unfolds to the base case, and the *else* side unfolds to the inductive case (Equation B.6).

$$\begin{aligned}
&\exists\ T_x.\\
&\ \ \mathit{comb}\ \mathit{sub}\ ((I_1,\ I_2,\ B),\ (T_x)\ \&\\
&\ \ \mathit{if}\ I_1 \geq I_2\\
&\ \ \mathit{then}\ \mathit{segComb}\ \mathit{add}\ T_x\ (T_n,\ \mathit{Bot})\\
&\ \ \mathit{else}\ \exists\ s_x,\ T_x.\\
&\ \ \ \ \ \ \mathit{segComb}\ \mathit{neg}\ (T_x)\ (T_x,\ s_x)\ \&\\
&\ \ \ \ \ \ \mathit{travComb}\ (s_x,\ T_x)\ (\mathit{add},\ (T_n,\ \mathit{Bot}))
\end{aligned} \tag{B.6}$$

There is no control circuitry for *add*. (All transactions transfer to *Bot*.) so the combinational representation of *add* in the *then* side of the *if–then–else* is just the combinational representation of the core. In the *else* side, we unfold using the inductive case. These transformations produce Equation B.7.

$$
\begin{array}{l}
\exists\ T_x. \\
\quad comb\ sub\ (I_1,\ I_2,\ B)\ T_x\ \& \\
\quad if\ I_1 \geq I_2 \\
\quad then\ comb\ add\ T_x\ T_n \\
\quad else\ \exists\ T_x. \\
\qquad comb\ neg\ T_x\ T_y\ \& \\
\qquad travComb\ (add,\ T_x)\ (add,\ (T_n,\ Bot))
\end{array} \tag{B.7}
$$

The last step in calculating the combinational representation of *diffAdd* is to unfold the one remaining *travComb* using the base case. As shown in Equations B.6 and B.7, the combinational representation of *add* is the combinational representation of *add* is the combinational representation of the core. This simplification produces Equation B.8.

$$
\begin{array}{l}
\exists\ T_x. \\
\quad comb\ sub\ (I_1,\ I_2,\ B)\ T_x\ \& \\
\quad if\ I_1 \geq I_2 \\
\quad then\ comb\ add\ T_x\ T_n \\
\quad else\ \exists\ T_x. \\
\qquad comb\ neg\ T_x\ T_y\ \& \\
\qquad comb\ add\ T_x\ T_n
\end{array} \tag{B.8}
$$

B.2 Traversing Through the Implementation of the DiffAdd Pipeline

In this section we illustrate the use of the relation *travPipe* to traverse through the implementation of a pipeline. To show how *travPipe* correctly represents the behavior of a pipeline, we work through several examples and demonstrate that *travPipe* is true for paths that are possible in the pipeline and false for impossible paths. By evaluating $travPipe\ (s_0,\ t_0)\ (s_n,\ t_n)$ we can calculate the set of all possible paths or determine that there are no possible paths connecting s_0 and s_n using t_0 as the entry time and t_n as the exit time. At each set in the evaluation we unfold *travPipe* using either the inductive case or the base case of the definition.

The *diffAdd* pipeline (Chapter 5, Figure B.1) calculates $|I_1 - I_2| + B$. There are three stages in the pipeline: S, N, and A. There are two paths through the pipeline. If $I_1 \geq I_2$, the transaction will follow the path from S to A (written $S \rightarrow A$). If $I_1 < I_2$, the transaction will follow the path $S \rightarrow N \rightarrow A$.

A transaction following the path $S \rightarrow A$ will stall for exactly one cycle in S if the previous transaction followed the path $S \rightarrow N \rightarrow A$. Transactions transferring from $S \rightarrow N$, $N \rightarrow A$, or $A \rightarrow Bot$ (where *Bot* is the bottom environment) will not stall. This results in three possible cases to test for a given latency.

1. $S \rightarrow N \rightarrow A$ Transactions spend one cycle in each stage.

2. $S \rightarrow A$ Transactions stall for one cycle in S. They spend two cycles in S and one in A.

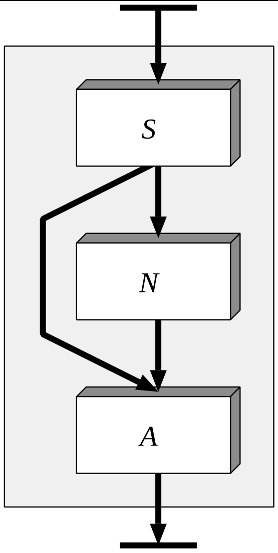


Figure B.1: *DiffAdd* pipeline

3. $S \rightarrow A$ Transactions do not stall, and so spend one cycle in each stage.

The matching relations for N and A are *matchDelay* 1, which means that transactions spend exactly one cycle in these stages. Transactions spend either one or two cycles in S. This information is captured in Equations B.9, B.10, and B.11.

$$\forall t_i,\ t_o\ .\ match\ S\ (t_i,\ t_o) \implies (t_o = t_i + 1)\ \vee\ (t_o = t_i + 2) \tag{B.9}$$
$$\forall t_i,\ t_o\ .\ match\ N\ (t_i,\ t_o) \implies (t_o = t_i + 1) \tag{B.10}$$
$$\forall t_i,\ t_o\ .\ match\ S\ (t_i,\ t_o) \implies (t_o = t_i + 1) \tag{B.11}$$

Examining the three possible cases, we see that transactions will spend either two or three clock cycles traversing the pipeline. This gives us two possible latencies to consider. The two possible latencies combined with the three possible paths results in a total of six cases to test, some of which are valid and some of which are invalid. We begin with a latency of two cycles (Equation B.12) and evaluate the three possible paths (Appendix B.2.1). In Appendix B.2.2 we begin with Equation B.13 describe the valid paths for a latency of three clock cycles. In Appendix B.2.3 we explain why latencies of one and for clock cycles do not contains any possible paths. Appendix C describes the use of *travPipe* with a pipeline that contains a loop (transactions may reuse a segment).

$$travPipe\ (S,\ t)\ (A,\ t{+}2) \tag{B.12}$$
$$travPipe\ (S,\ t)\ (A,\ t{+}3) \tag{B.13}$$

B.2.1 Traversing DiffAdd in Two Clock Cycles

To evaluate the possible paths for a latency of two cycles, we begin with Equation B.12. The segments (S and A) are different, so we unfold *travPipe* using the inductive case, resulting in the call to *travPipe* in Equation B.14.

$$\begin{array}{l} \exists\ s_x,\ t_x\ . \\ \quad match\ S\ (t,\ t_x)\ \& \\ \quad xfr\ (S,\ s_x)\ t_x\ \& \\ \quad travPipe\ (s_x,\ t_x)\ (A,\ t{+}2) \end{array} \tag{B.14}$$

We now need to find instantiations for s_x (the next stage to transfer to) and t_x (the time of the transfer). The transaction will transfer to either A or N, based on whether $I_1 \geq I_2$. So, we have two cases to test:

1. $I_1 \geq I_2$ the transaction transfers from S to A, shown in Equation B.15
2. $I_1 < I_2$ the transaction transfers from S to N, shown in Equation B.20

In the case where $I_1 \geq I_2$, we begin with Equation B.14 and instantiate s_x with A, resulting in Equation B.15.

$$\begin{array}{l} \exists\ t_x\ . \\ \quad match\ S\ (t,\ t_x)\ \& \\ \quad xfr\ (S,\ A)\ t_x\ \& \\ \quad travPipe\ (A,\ t_x)\ (A,\ t{+}2) \end{array} \tag{B.15}$$

The transfer from S to A might stall, so we do a case split on whether there is a stall. If there is not a stall, the transaction will remain in S for one cycle; otherwise it will remain for two cycles. If there is no stall (Equation B.16) we instantiate t_x with t+1 and if there is a stall (Equation B.18) we instantiate t_x with t+2.

$$\begin{array}{l} match\ S\ (t,\ t{+}1)\ \& \\ xfr\ (S,\ A)\ (t{+}1)\ \& \\ travPipe\ (A,\ t{+}1)\ (A,\ t{+}2) \end{array} \tag{B.16}$$

We choose the base case of *travPipe* (Equation B.17), because the two segments in the call to *travPipe* in Equation B.16 are both A.

$$\begin{array}{l} match\ S\ (t,\ t{+}1)\ \& \\ xfr\ (S,\ A)\ (t{+}1)\ \& \\ (A = A)\ \&\ match\ A\ (t{+}1,\ t{+}2) \end{array} \tag{B.17}$$

Equation B.17 describes a valid path, because the matching relations describe valid latencies for the segments as described in Equations B.9–B.11.

We now consider the case where the transaction stalls before transferring from S to A. This is the second case resulting from Equation B.15. We instantiate t_x with t+2, resulting in Equation B.18.

$$\begin{array}{l} match\ S\ (t,\ t{+}2)\ \& \\ xfr\ (S,\ A)\ (t{+}2)\ \& \\ travPipe\ (A,\ t{+}2)\ (A,\ t{+}2) \end{array} \tag{B.18}$$

In Equation B.18, the segments in the call to *travPipe* are both A, so we use the base case of *travPipe*, producing Equation B.19.

match S $(t,\ t{+}2)$ &
xfr $(S,\ A)$ $(t{+}2)$ &
$(A = A)$ & *match* A $(t{+}2,\ t{+}2)$ (B.19)

Equation B.19 does not describe a valid path through *diffAdd*. In Equation B.19, the transaction supposedly enters and exits A at time $t{+}2$, but this violates Equation B.11, which says that transactions spend exactly one clock cycle in A.

This completes the analysis of the two possible paths (with and without a stall in the transfer between S and A) through the pipeline for transactions with $I_1 \geq I_2$ and a total latency of two cycles. We now examine the second case from Equation B.12: transactions with $I_1 < I_2$, which follow the path $S{\rightarrow}N{\rightarrow}A$.

From Equation B.12, we instantiate s_x with N and t_x with $t{+}1$. Because the transfer from S to N cannot stall, $t{+}1$ is the only possible choice for instantiating t_x. This results in Equation B.20.

match S $(t,\ t{+}1)$ &
xfr $(S,\ N)$ $(t{+}1)$ &
travPipe $(N,\ t{+}1)$ $(A,\ t{+}2)$ (B.20)

We choose the inductive case of *travPipe* in Equation B.20 to produce Equation B.21.

match S $(t,\ t{+}1)$ &
xfr $(S,\ N)$ $(t{+}1)$ &
$\exists\ s_x,\ t_x$.
 match N $(t{+}1,\ t_x)$ &
 xfr $(N,\ s_x)$ t_x &
 travPipe $(A,\ t_x)$ $(A,\ t{+}2)$ (B.21)

We instantiate s_x with A. The transfer from N to A cannot stall so we instantiate t_x with $t{+}2$, resulting in Equation B.22.

match S $(t,\ t{+}1)$ &
xfr $(S,\ N)$ $(t{+}1)$ &
match N $(t{+}1,\ t{+}2)$ &
xfr $(N,\ A)$ $(t{+}2)$ &
travPipe $(A,\ t{+}2)$ $(A,\ t{+}2)$ (B.22)

From the evaluation of Equation B.19 we know that *travPipe* $(A,\ t{+}2)$ $(A,\ t{+}2)$ is not valid, and so Equation B.22 does not describe a valid path.

This completes the analysis of traversing *diffAdd* with a latency of two clock cycles. The paths resulting from the three different cases are shown in Equations B.17, B.19, and B.22. Of the three cases, only Equation B.17 (the case with $I_1{\geq}I_2$ and no stall) describes a valid path.

B.2.2 Traversing DiffAdd in Three Clock Cycles

The pipeline in *diffAdd* has valid paths with latencies of either two or three clock cycles. Evaluating the three possible paths using a latency of three clock cycles is very similar to the two cycle analysis. The path $S \rightarrow N \rightarrow A$, which is taken by transactions with $(I_1 < I_2)$, is valid with a latency of three clock cycles. The path $S \rightarrow A$ is also valid, if the transaction stalls before the transfer from S to A. The third possible path ($S \rightarrow A$ with no stall) is not valid, and we include its evaluation here to demonstrate what happens when the specified latency is longer than that of a valid path.

We begin with Equation B.23, which is a copy of Equation B.13 and which unfolds to Equation B.24.

$$\mathit{travPipe}\ (S,\ t)\ (A,\ t{+}3) \tag{B.23}$$

$$\begin{aligned}&\exists\ s_x,\ t_x\,.\\ &\ \mathit{match}\ S\ (t,\ t_x)\ \&\\ &\ \mathit{xfr}\ (S,\ s_x)\ t_x\ \&\\ &\ \mathit{travPipe}\ (s_x,\ t_x)\ (A,\ t{+}3)\end{aligned} \tag{B.24}$$

The next stage is A and we are considering the case where there is no stall, so we instantiate s_x with A and t_x with $t{+}1$ (Equation B.25)

$$\begin{aligned}&\mathit{match}\ S\ (t,\ t{+}1)\ \&\\ &\mathit{xfr}\ (S,\ A)\ (t{+}1)\ \&\\ &\mathit{travPipe}\ (A,\ t{+}1)\ (A,\ t{+}3)\end{aligned} \tag{B.25}$$

If we use the base case of *travPipe*, we end up with $\mathit{match}\ A\ (t{+}1,\ t{+}3)$. This is false, because Equation B.11 says that A has a latency of exactly one cycle, and we have a latency of two cycles (enter at $t{+}1$ and exit at $t{+}3$). If we use the inductive case, we must find a intermediate segment s_x that satisfies Equation B.26.

$$\begin{aligned}&\mathit{match}\ s\ (t,\ t{+}1)\ \&\\ &\mathit{xfr}\ (S,\ A)\ (t{+}1)\ \&\\ &\exists\ s_x,\ t_x\,.\\ &\ \mathit{match}\ A\ (t{+}1,\ t_x)\ \&\\ &\ \mathit{xfr}\ (A,\ s_x)\ t_x\ \&\\ &\ \mathit{travPipe}\ (s_x,\ t_x)\ (A,\ t{+}3)\end{aligned} \tag{B.26}$$

There is no segment s_x that satisfies Equation B.26, because transactions from A can only exit the pipeline. (Transfer to *Bot*.) Transactions can only transfer in to *Bot*, they can not transfer out from *Bot*. This means that $\mathit{travPipe}\ (\mathit{Bot},\ t_x)\ (A,\ t{+}3)$ is false. Thus, using either the base or inductive case of *travPipe* in Equation B.25 results in an invalid path, and so Equation B.25 does not describe any valid paths.

B.2.3 Traversing DiffAdd with Invalid Latencies

The *diffAdd* pipeline has valid latencies of two and three clock cycles. If we try to find a valid path with a latency of one cycle (Equation B.27), we end up with Equations B.28 or B.29 depending on whether the transactions stall in S.

$$\mathit{travPipe}\ (S,\ t)\ (A,\ t{+}1) \tag{B.27}$$

$$\begin{aligned}
&\exists\ t_x.\\
&\ \mathit{match}\ S\ t\ (t{+}1)\ \&\\
&\ \mathit{xfr}\ (S,\ s_x)\ (t{+}1)\ \&\\
&\ \mathit{travPipe}\ (s_x,\ t{+}1)\ (A,\ t{+}1)
\end{aligned} \tag{B.28}$$

$$\begin{aligned}
&\exists\ s_x.\\
&\ \mathit{match}\ S\ t\ (t{+}2)\ \&\\
&\ \mathit{xfr}\ (S,\ s_n)\ (t{+}2)\ \&\\
&\ \mathit{travPipe}\ (s_x,\ t{+}2)\ (A,\ t{+}1)
\end{aligned} \tag{B.29}$$

In both Equations B.28 and B.29, the call to *travPipe* ($\mathit{travPipe}\ (s_x{,}t{+}1)\ (A{,}t{+}1)$) and $\mathit{travPipe}\ (s_x,\ t{+}2)\ (A,\ t{+}1)$) is false, because the latency for the traversal is either zero clock cycles (Equation B.28) or negative (Equation B.29).

Appendix C

Traversing the Implementation of a Pipeline with Loops

In this appendix we illustrate the use of *travPipe* with a pipeline that contains a loop (transactions reuse a segment). In particular, we demonstrate how we detect if a transaction remains within a segment for some period of time (t_0 to t_n) or if the transaction enters the segment at t_0, exits, and reenters, and then exits a second time at t_n. The *diffAdd* pipeline does not contain any loops; in this section we use a pipeline with a similar topology and a loop. The pipeline (*maf*, for multiply/add floating point) contains three stages: A (addition), M (multiply), and N (normalize) and supports two instructions: multiply and add. Addition transactions use A and then N. Multiply transactions use A, then M once or twice and then N. The conditional reuse of M is dependent upon the magnitude of the first operand. A multiply transaction with operands a and b will use M once if $a < max/2$ and twice if $a \geq max/2$, where max is the maximum value of an operand. This is an example of data-dependent control.

In *diffAdd*, transactions that skip N (transfer from S to A) might stall for one cycle in S. Similarly, in *maf*, transactions that skip M might stall for one cycle in A before transferring to N. None of the other possible transfers ($A{\rightarrow}M$, $M{\rightarrow}M$, $M{\rightarrow}N$, or $N{\rightarrow}Bot$) can stall. The latency for A is one or two clock cycles (Equation C.1) and the latency of M and N is one (Equations C.2 and C.3).

$$\forall t_i,\ t_o\ .\ match\ A\ (t_i,\ t_o) \implies (t_o = t_i + 1\ \vee\ t_o = t_i + 2) \tag{C.1}$$

$$\forall t_i,\ t_o\ .\ match\ M\ (t_i,\ t_o) \implies (t_o = t_i + 1) \tag{C.2}$$

$$\forall t_i,\ t_o\ .\ match\ N\ (t_i,\ t_o) \implies (t_o = t_i + 1) \tag{C.3}$$

We examine the multiply instruction and use a latency of four clock cycles (t to t+4) in Equation C.4. Building the two possible paths for a multiply (with and without the reuse of M), results in Equations C.5 and C.6 respectively.

$$travPipe\ (A,\ t)\ (N,\ t{+}4) \tag{C.4}$$

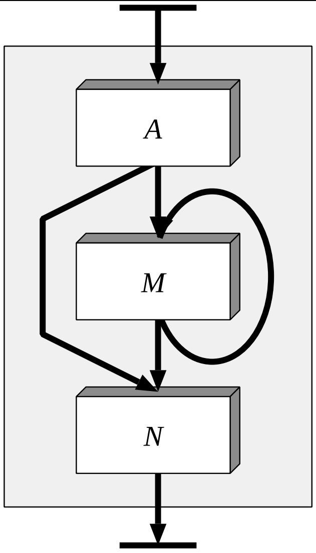


Figure C.1: *Maf* pipeline

$$\begin{array}{l} match\ A\ (t,\ t{+}1)\ \& \\ xfr\ (A,\ M)\ t{+}1\ \& \\ match\ M\ (t{+}1,\ t{+}2)\ \& \\ xfr\ (M,\ M)\ t{+}2\ \& \\ match\ M\ (t{+}2,\ t{+}3)\ \& \\ xfr\ (M,\ N)\ (t{+}3)\ \& \\ match\ N\ (t{+}3,\ t{+}4) \end{array} \tag{C.5}$$

In Equation C.5, the transaction uses each stage for exactly one cycle and so meets the matching relations for all three segments. Thus, Equation C.5 describes a valid path.

$$\begin{array}{l} match\ A\ (t,\ t{+}1)\ \& \\ xfr\ (A,\ M)\ t{+}1\ \& \\ match\ M\ (t{+}1,\ t{+}3)\ \& \\ xfr\ (M,\ N)\ (t{+}3)\ \& \\ match\ N\ (t{+}3,\ t{+}4) \end{array} \tag{C.6}$$

In Equation C.5, the transactions use M twice, once from t+1 to t+2 and once from t+2 to t+3. In Equation C.6, the transaction remains in M from t+1 to t+3. The latency for M is one, (Equation C.2) thus $match\ M\ (t{+}1,\ t{+}3)$ is false and Equation C.6 does not describe a valid path. By using the matching relations for the segments we were able to determine that if a multiply transaction has a latency of four clock cycles, then it used M twice. This completes the description of traversing the implementation of a pipeline using *travPipe*.

Appendix D

Proofs

D.1 Proof of Induction Lemma for TermComb (Lemma 8.8)

$\vdash$ $termComb\ (s_0,\ T_0)\ \iff$
$(\exists\ T_n\ .\ (segComb[s_0]\ T_0\ (T_n,\ Bot))\ \ \&\ \ (s_0 = Exit))$
$\vee$
$\exists\ s_x,\ T_x\ .$
$segComb[s_0]\ T_0\ (T_x,\ s_x)\ \&$
$termComb\ (s_x,\ T_x)$

BY `Unfold` *termComb*

$\vdash$ $\exists T_n\ .\ travComb\ (s_0,\ T_0)\ (Exit,\ (T_n,\ Bot))$
$\iff \ldots$

BY `Unfold` *travComb* `(*` Definition 7.18 `*)`

$\vdash$ $(\exists\ T_n\ .$
$(s_0 = Exit)\ \ \&\ (segComb[s_0]\ T_0\ (T_n,\ Bot))$
$\vee$
$\exists\ s_x,\ T_x\ .$
$segComb[s_0]\ T_0\ (T_x,\ s_x)\ \&$
$travComb\ (s_x,\ T_x)\ (Exit,\ (T_n,\ Bot))$
$\iff \ldots$

BY `Push` $\exists T_n$ `down into concl`

$\vdash$ $(\exists\ T_n\ .\ s_0 = Exit\ segComb[s_0]\ T_0\ (T_n,\ Bot)$
$\vee$
$\exists\ s_x,\ T_x\ .$
$segComb[s_0]\ T_0\ (T_x,\ s_x)\ \&$
$\exists\ T_n\ .\ travComb\ (s_x,\ T_x)\ (Exit,\ (T_n,\ Bot))$
$\iff \ldots$

BY `Fold` *termComb* `(*` Definition 8.4 `*)`

$\vdash$ $(\exists\ T_n\ .$
$segComb[s_0]\ T_0\ (T_n,\ Bot)\ \&\ s_0 = Exit)$
$\vee$
$\exists\ s_x,\ T_x\ .$
$segComb[s_0]\ T_0\ (T_x,\ s_x)\ \&$
$termComb\ (s_x,\ T_x)$
$\iff \ldots$

■

D.2 Proof of Lemma for Termination in Implementation (Lemma 8.19)

$\vdash$ $\forall t_0$. $termPipe\ (Entry,\ t_0) \implies$
$XfrIn\ t_0 \implies \exists!\ t_n\ .\ Match\ I\ O\ t_0\ t_n$

BY `Begin proof`

- $xfrIn\ (Entry,\ t_0) \implies$
  $\exists!\ t_n\ .\ travPipe\ (Entry,\ t_0)\ (Exit,\ t_n)\ \&\ xfr\ (Exit,\ Bot)\ t_n)$

$\vdash$ $xfrIn\ (Entry,\ t_0) \implies \exists!\ t_n\ .\ Match\ I\ O\ t_0\ t_n$

BY `Unfold` *termPipe*`,` *XfrIn* `(*` Definitions 8.7, 8.6 `*)`

- $xfrIn\ (Entry,\ t_0) \implies$
  $\exists!\ t_n\ .\ travPipe\ (Entry,\ t_0)\ (Exit,\ t_n)\ \&\ xfr\ (Exit,\ Bot)\ t_n)$

$\vdash$ $\forall\ t_0\ .\ xfrIn\ (Entry,\ t_0) \implies \exists!\ t_n\ .\ Match\ I\ O\ t_0\ t_n$

BY `Specification 7.18 (*` $travPipe \implies Match$ `*)`

- $xfrIn\ (Entry,\ t_0) \implies$
  $\exists!\ t_n\ .\ travPipe\ (Entry,\ t_0)\ (Exit,\ t_n)\ \&\ xfr\ (Exit,\ Bot)\ t_n)$

$\vdash$ $xfrIn\ (Entry,\ t_0) \implies$
$\exists!\ t_n\ .\ travPipe\ (Entry,\ t_0)\ (Exit,\ t_n)\ \&\ xfr\ (Exit,\ Bot)\ t_n)$

■

D.3 Proof of Termination in Combinational Representation (Lemma 8.18)

$\vdash \ \forall \ T_0 \ . \ termComb(Entry, \ T_0)$

BY `Specification 7.13`

- $\forall \ T_0 \ . \ \exists T_n \ . \ travComb \ (Entry, \ T_0) \ (Exit, \ (T_n, \ Bot))$

$\vdash \ \forall \ T_0 \ . \ termComb(Entry, \ T_0)$

BY `Unfold` *termComb* `(*` Definition 8.4 `*)`

- $\forall \ T_0 \ . \ \exists T_n \ . \ travComb \ (Entry, \ T_0) \ (Exit, \ (T_n, \ Bot))$

$\vdash \ \forall \ T_0 \ . \ \exists T_n \ . \ travComb \ (Entry, \ T_0) \ (Exit, \ (T_n, \ Bot))$

■

D.4 Proof That Every Request Is Eventually Accepted (Theorem 8.5)

Lemma D.1: *Every request eventually becomes highest priority request*
$\vdash \ \forall \ s_0, \ s_1, \ t_r \ . \ \exists \ t_p \ . \ holdUntil \ (req \ (s_0, \ s_1)) \ t_r \ (priReq \ (s_0, \ s_1)) \ t_p$

Lemma D.2: *Hold until at last time*
$\vdash \ \forall \ \mathcal{P}_0, \ \mathcal{P}_1, \ t_0, \ t_1 \ .$
$$(\mathcal{P}_0 \ t_0) \ \& \ (holdUntil \ \mathcal{P}_0 \ t_0 \ \mathcal{P}_1 \ t_1) \implies (\mathcal{P}_0 \ t_0 \ \& \ \mathcal{P}_1 \ t_1)$$

Lemma D.3: *Transitivity of holdUntil*

$$
\begin{array}{l}
\vdash \ \forall\ \mathcal{P}_0,\ \mathcal{P}_1,\ \mathcal{P}_2\ . \\
\qquad (\forall\ t\ .\ \mathcal{P}_2\ t \implies \mathcal{P}_1\ t)\ \& \\
\qquad (\forall\ t\ .\ \mathcal{P}_1\ t \implies \mathcal{P}_0\ t) \implies \\
\qquad\quad \forall\ t_0,\ t_1,\ t_2\ . \\
\qquad\qquad \mathit{holdUntil}\ \mathcal{P}_0\ t_0\ \mathcal{P}_1\ t_1\ \& \\
\qquad\qquad \mathit{holdUntil}\ \mathcal{P}_1\ t_1\ \mathcal{P}_2\ t_2 \implies \\
\qquad\qquad\quad \mathit{holdUntil}\ \mathcal{P}_0\ t_0\ \mathcal{P}_2\ t_2
\end{array}
$$

Lemma D.4: *Accept highest priority request*

$$\vdash\ \forall\ s_0,\ s_1\ .\ \forall\ t\ .\ \mathit{acc}\ (s_0,\ s_1)\ t \implies ((\mathit{req}\ (s_0,\ s_1)\ t)\ \&\ (\mathit{prev}\ s_1\ t = s_0))$$

$\vdash \quad \forall\ s_0,\ s_1,\ t_r\ .\ req\ (s_0,\ s_1)\ t_r \quad \Longrightarrow$
$\quad \exists\ t_a\ .$
$\quad\quad holdUntil\ (req\ (s_0,\ s_1))\ t_r\ (acc\ (s_0,\ s_1))\ t_a$

BY `Begin proof`

- $req\ (s_0,\ s_1)\ t_r$

$\vdash \quad \exists t_a\ .$
$\quad holdUntil\ (req\ (s_0,\ s_1))\ t_r\ (acc\ (s_0,\ s_1))\ t_a$

BY `Lemma 7.1 (*` $\exists t_1\ .\ holdUntil\ \mathcal{P}_0\ t_0\ \mathcal{P}_1\ t_1$ `*)`
`with` $\mathcal{P}_0\ t = req\ (s_0,\ s_1)\ t$, $\quad \mathcal{P}_1\ t = (prev\ (s_0,\ s_1)\ t)$, $\quad t_0 = t_r$

$\vdash \quad \forall t\ .$
$\quad req\ (s_0,\ s_1)\ t\ \&\ \neg(acc\ (s_0,\ s_1)\ t) \quad \Longrightarrow$
$\quad\quad req\ (s_0,\ s_1)\ (t{+}1)$

BY `Lemma 8.1`

■

$\vdash \quad \exists t_1\ .\ (\forall t'\ :\ \{t_0..t_1\}\ .\ req\ (s_0,\ s_1)\ t') \Longrightarrow prev\ (s_0,\ s_1)\ t_1$

BY `Lemma D.1`

■

- $req\ (s_0,\ s_1)\ t_r$
- $\exists t_p\ .\ holdUntil\ (req\ (s_0,\ s_1))\ t_r\ (prev\ (s_0,\ s_1))\ t_p$

(D.2) $\vdash \quad \exists t_a\ .\ holdUntil\ (req\ (s_0,\ s_1))\ t_r\ (acc\ (s_0,\ s_1))\ t_a$

Figure D.1: Beginning of proof of Theorem 8.5

$\vdash \quad \exists t_a \;.\; holdUntil\;(req\;(s_0,\;s_1))\;t_r\;(acc\;(s_0,\;s_1))\;t_a$

BY `Lemma 7.1 (*` $\exists t_1 \;.\; holdUntil\;\mathcal{P}_0\;t_0\;\mathcal{P}_1\;t_1$ `*)`
`with` $\mathcal{P}_0\;t = priReq\;(s_0,\;s_1)\;t),$
$\mathcal{P}_1\;t = acc\;(s_0,\;s_1)\;t,\;\; t_0 = t_p$

$\vdash \quad \forall\;t\;.\;priReq\;(s_0,\;s_1)\;t\;\;\&\;\;\neg(acc\;(s_0,\;s_1)\;t)\quad\Longrightarrow$
$((req\;(s_0,\;s_1)\;(t{+}1))\;\&\;(prev\;s_1\;(t{+}1)) = s_0)$

$\vdash \quad \forall\;t_0\;.$
$\exists\;t_1\;.\;holdCmmd\;(priReq\;(s_0,\;s_1))\;t_0\;(acc\;(s_0,\;s_1))\;t_1$

$\vdash \quad priReq\;(s_0,\;s_1)\;t_p$

BY `Lemma D.2`
`(*` $holdUntil\;\mathcal{P}_0\;t_0\;\mathcal{P}_1\;t_1\quad\Longrightarrow\quad\mathcal{P}_0\;t_0\;\;\&\;\;\mathcal{P}_1\;t_1$ `*)`

■

- $req\;(s_0,\;s_1)\;t_r$
- $holdUntil\;(req\;(s_0,\;s_1))\;t_r\;(prev\;(s_1,\;s_0))\;t_p$
- $holdUntil\;(priReq\;(s_0,\;s_1))\;t_p\;(acc\;(s_0,\;s_1))\;t_a$

(D.3) $\vdash \quad \exists t_a\;.$
$holdUntil\;(req\;(s_0,\;s_1))\;t_r\;(acc\;(s_0,\;s_1))\;t_a$

Figure D.2: Proof step in Theorem 8.5; every request eventually becomes highest priority request

$\vdash \quad \exists t_a \ . \ \mathit{holdUntil} \ (\mathit{req} \ (s_0, \ s_1)) \ t_r \ (\mathit{acc} \ (s_0, \ s_1)) \ t_a$

BY `Instantiate concl with` $t_a = t_a$

- $\mathit{holdUntil} \ (\mathit{req} \ (s_0, \ s_1)) \ t_r \ (\mathit{prev} \ (s_0, \ s_1)) \ t_p$
- $\mathit{holdUntil} \ (\mathit{priReq} \ (s_0, \ s_1)) \ t_p \ (\mathit{acc} \ (s_0, \ s_1)) \ t_a$

$\vdash \quad \mathit{holdUntil} \ (\mathit{req} \ (s_0, \ s_1)) \ t_r \ (\mathit{acc} \ (s_0, \ s_1)) \ t_a$

BY `Lemma D.3 (*` transitivity of *holdUntil* `*)`

$\vdash \quad \forall \ t \ . \ \mathit{acc}(s_0, \ s_1) \ t \implies (\mathit{priReq} \ (s_0, \ s_1) \ t)$

BY `Lemma D.4`

$\blacksquare$

$\vdash \quad \forall \ t \ . \ (\mathit{priReq} \ (s_0, \ s_1) \ t) \implies \mathit{req} \ (s_0, \ s_1) \ t$

$\blacksquare$

$\vdash \quad \mathit{holdUntil} \ (\mathit{req} \ (s_0, \ s_1)) \ t_r \ (\mathit{acc} \ (s_0, \ s_1)) \ t_a$

$\blacksquare$

Figure D.3: Proof step for Theorem 8.5; use transitivity of *holdUntil*

D.5 Proof of Correctness of Transfers (Lemma 8.14)

Lemma D.5: *Transfer in implies match*

$\vdash \forall s_0,\ s_1,\ t_0,\ T\ .$
$\quad xfrIn\ s_0\ t_0 \implies$
$\quad\quad \exists!\ t_1\ .\ match\ s\ t_0\ t_1$

Lemma D.6: *Segments have equal inputs for equal outputs*

$\vdash \forall s_0,\ T_0,\ s_1,\ s_1',\ T_1,\ T_1'\ .$
$\quad segComb\ s_0\ T_0\ (T_1,\ s_1)\ \&$
$\quad segComb\ s_0\ T_0\ (T_1',\ s_1') \implies$
$\quad\quad (T_1\ =\ T_1')\ \&\ (s_1\ =\ s_1')$

Lemma D.7: *Correctness of the path between segments (Copy of Lemma 8.14)*

$\vdash \forall s_0,\ s_1,\ t_0,\ T\ .$
$\quad xfrIn\ s_0\ t_0\ \&$
$\quad segComb\ s_0\ (transP\ s_0\ t_0)\ (T,\ s_1) \implies$
$\quad\quad \exists!\ t_1\ .\ match\ s_0\ t_0\ t_1\ \&$
$\quad\quad\quad xfr\ (s_0,\ s_1)\ t_1$

D.6 Proof of Forward Direction of One-to-One (Theorem 8.7)

$\vdash \quad \forall t_0 \ . \ XfrIn\ t_0 \implies \exists!\ t_n \ . \ Match\ I\ O\ t_0\ t_n$

BY `Lemma 8.18 (*` $\forall T_0 \ . \ termComb\ (Entry,\ T_0)$ `*)`

- $\forall\ T_0 \ . \ termComb\ (Entry,\ T_0)$

$\vdash \quad \forall t_0 \ . \ XfrIn\ t_0 \implies \exists!\ t_n \ . \ Match\ I\ O\ t_0\ t_n$

BY `Lemma 8.19 (*` $termPipe \implies (XfrIn \implies Match)$ `*)`

- $\forall\ T_0 \ . \ termComb\ (Entry,\ T_0)$

(D.5) $\vdash \ \forall\ t_0 \ . \ termPipe\ (Entry,\ t_0)$

Figure D.4: Theorem 8.7 — One-to-one forward; first step

- $\forall\ T_0\ .\ termComb\ (Entry,\ T_0)$

$\vdash\ \forall\ t_0\ .\ termPipe\ (Entry,\ t_0)$

BY `Lemma 8.8 with` $\mathcal{P}\ s_0\ T_0 = \forall\ t_0\ .\ termPipe\ (s_0,\ t_0)$
`(*` Induction lemma for *termComb* `*)`
`THEN Lemma 8.20 (*` *termPipe* can be treated inductively `*)`

- $\forall\ T_0\ .$
  $Entry{=}Exit\ \ \&$
  $\exists\, T_n\ .\ segComb\ Entry\ T_0\ (Exit,\ (T_n,\ Bot))$

$$\text{(D.6)} \vdash\ xfrIn\ Entry\ t_0\ \ \Longrightarrow$$
$$\exists!\ t_n\ .$$
$$(Entry = Exit)\ \ \&\ \ (match\ Entry\ (t_0,\ t_n))\ \ \&$$
$$(xfr\ (Exit,\ Bot)\ t_n$$

- $\forall\ T_0\ .$
  $\exists\ s_x,\ T_x\ .$
  $(segComb\ Entry\ T_0\ (T_x,\ s_x))\ \ \&$
  $(\forall\ t_x\ .\ termPipe\ (s_x,\ t_x))$

$$\text{(D.8)} \vdash\ xfrIn\ Entry\ t_0\ \ \Longrightarrow$$
$$\exists!\ s_x,\ t_x\ .$$
$$(match\ Entry\ (t_0,\ t_x))\ \ \&$$
$$(xfr\ (Entry,\ s_x)\ t_x)\ \ \&$$
$$(termPipe\ (s_x,\ t_x))$$

Figure D.5: Theorem 8.7 — One-to-one forward; perform induction on *termComb*

- $\forall\ T_0\ .\ \mathit{Entry}{=}\mathit{Exit}\ \ \&\ \ \exists\, T_n\ .\ \mathit{segComb}\ \mathit{Entry}\ T_0\ (\mathit{Exit},\ (T_n,\ \mathit{Bot}))$

$\vdash\ \ \forall\ t_0\ .\ \mathit{xfrIn}\ \mathit{Entry}\ t_0\ \ \Longrightarrow$
$\quad \exists!\ t_n\ .$
$\qquad \mathit{Entry} = \mathit{Exit}\ \&$
$\qquad \mathit{match}\ \mathit{Entry}\ (t_0,\ t_n)\ \&$
$\qquad \mathit{xfr}\ (\mathit{Exit},\ \mathit{Bot})\ t_n$

BY `Instantiate hyp with` $T_0{=}\mathit{transP}\ \mathit{Entry}\ t_0$

- $\mathit{Entry}{=}\mathit{Exit}$
- $\mathit{segComb}\ \mathit{Entry}\ T_0\ (\mathit{Exit},\ (T_n,\ \mathit{Bot}))$
- $\mathit{xfrIn}\ \mathit{Entry}\ t_0$

$\vdash\ \ \exists!\ t_n\ .$
$\quad \mathit{Entry} = \mathit{Exit}\ \&$
$\quad \mathit{match}\ \mathit{Entry}\ (t_0,\ t_n)\ \&$
$\quad \mathit{xfr}\ (\mathit{Exit},\ \mathit{Bot})\ t_n$

BY `Lemma 8.14`
(* Transfer into a segment implies *)
(* transfer to next correct segment *)

- $\mathit{Entry}{=}\mathit{Exit}$
- $\mathit{match}\ \mathit{Entry}\ t_0\ t_x$
- $\mathit{xfr}\ (\mathit{Entry},\ \mathit{Bot})\ t_x$

(D.7) $\vdash\ \ \exists!\ t_n\ .$
$\quad \mathit{Entry} = \mathit{Exit}\ \&$
$\quad \mathit{match}\ \mathit{Entry}\ (t_0,\ t_n)\ \&$
$\quad \mathit{xfr}\ (\mathit{Exit},\ \mathit{Bot})\ t_n$

Figure D.6: Theorem 8.7 — One-to-one forward; base case – 1

- $Entry{=}Exit$
- $match\ Entry\ t_0\ t_x$
- $xfr\ (Entry,\ Bot)\ t_x$

$\vdash$ $\exists!\ t_n\ .$
$Entry = Exit$ &
$match\ Entry\ (t_0,\ t_n)$ &
$xfr\ (Exit,\ Bot)\ t_n$

BY `Instantiate concl with` $t_n{=}t_x$

- $Entry{=}Exit$
- $match\ Entry\ t_0\ t_x$
- $xfr\ (Entry,\ Bot)\ t_x$

$\vdash$ $Entry = Exit$
■

$\vdash$ $match\ Entry\ (t_0,\ t_x)$
■

$\vdash$ $xfr\ (Exit,\ Bot)\ t_x$
■

Figure D.7: Theorem 8.7 — One-to-one forward; base case – 2

- $\forall\ T_0\ .\ \exists s_x,\ T_x\ .$
  $segComb\ Entry\ T_0\ (T_x,\ s_x)$ &
  $\forall\ t_x\ .\ termPipe\ (s_x,\ t_x)$

$\vdash$ $xfrIn\ Entry\ t_0 \implies$
$\exists!\ s_x,\ t_x\ .$
$(match\ Entry\ t_0\ t_x)$ & $(xfr\ (Entry,\ s_x)\ t_x)$ & $(termPipe\ (s_x,\ t_x)$

BY `Instantiate hyp with` T_0=($transP\ Entry\ t_0$)

- $xfrIn\ Entry\ t_0$
- $segComb\ Entry\ (transP\ Entry\ t_0)\ (T_x,\ s_x)$
- $\forall\ t_x\ .\ termPipe\ (s_x,\ t_x)$

$\vdash$ $\exists!\ s_x,\ t_x\ .$
$(match\ Entry\ t_0\ t_x)$ & $(xfr\ (Entry,\ s_x)\ t_x)$ & $(termPipe\ (s_x,\ t_x))$

BY `Lemma 8.14`
(* Transfer into a segment implies *)
(* transfer to correct next segment *)

- $\exists!\ t_x\ .\ (match\ Entry\ t_0\ t_x)$ & $(xfr\ (Entry,\ s_x)\ t_x)$
- $\forall t_x\ .\ termPipe\ (s_x,\ t_x)$

(D.9) $\vdash$ $\exists!\ s_x,\ t_x\ .$
$(match\ Entry\ t_0\ t_x)$ & $(xfr\ (Entry,\ s_x)\ t_x)$ & $(termPipe\ (s_x,\ t_x))$

Figure D.8: Theorem 8.7 — One-to-one forward; inductive case

- $\exists!\ t_x\ .\ (match\ Entry\ t_0\ t_x)\ \&\ (xfr\ (Entry,\ s_x)\ t_x)$
- $\forall t_x\ .\ termPipe\ (s_x,\ t_x)$

$\vdash\ \exists!\ s_x,\ t_x\ .$
$(match\ Entry\ t_0\ t_x)\ \ \&\ \ (xfr\ (Entry,\ s_x)\ t_x)\ \ \&\ \ (termPipe\ (s_x,\ t_x))$

BY `Lemma 8.16 (*` split goal for unique existential quantifier `*)`
`with` $\mathcal{P}_0\ (s,\ t) = (match\ Entry\ t_0\ t)\ \&\ (xfr\ (Entry,\ s)\ t)$,
$\mathcal{P}_1\ (s,\ t) = termPipe\ (s,\ t)$

- $(match\ Entry\ t_0\ t_x)\ \&\ (xfr\ (Entry,\ s_x)\ t_x)$
- $\forall t_x\ .\ termPipe\ (s_x,\ t_x)$

$\vdash\ \exists\ s_x,\ t_x\ .$
$(match\ Entry\ t_0\ t_x)\ \ \&\ \ (xfr\ (Entry,\ s_x)\ t_x)\ \ \&$
$(termPipe\ (s_x,\ t_x))$

BY `Instantiate concl with` $s_x{=}s_x$`,` $t_x{=}t_x$
`THEN Instantiate hyp with` $t_x{=}t_x$

- $(match\ Entry\ t_0\ t_x)\ \ \&\ \ (xfr\ (Entry,\ s_x)\ t_x)$
- $termPipe\ (s_x,\ t_x)$

$\vdash\ (match\ Entry\ t_0\ t_x)\ \&\ (xfr\ (Entry,\ s_x)\ t_x)\ \&$
$(termPipe\ (s_x,\ t_x))$

■

- $\exists!\ t_x\ .\ (match\ Entry\ t_0\ t_x)\ \&\ (xfr\ (Entry,\ s_x)\ t_x)$

$\vdash\ \exists!\ s_x,\ t_x\ .\ (match\ Entry\ t_0\ t_x)\ \ \&\ \ (xfr\ (Entry,\ s_x)\ t_x)$

BY `Lemma 8.21`
`(*` $(\exists\ s_x\ .\ xfr\ (Entry,\ s_x)\ t_x)\ \ \Longrightarrow$ `*)`
`(*` $(\exists!\ s_x\ .\ xfr\ (Entry,\ s_x)\ t_x)$ `*)`

- $\exists!\ s_x,\ t_x\ .\ (match\ Entry\ t_0\ t_x)\ \ \&\ \ (xfr\ (Entry,\ s_x)\ t_x)$

$\vdash\ \exists!\ s_x,\ t_x\ .\ (match\ Entry\ t_0\ t_x)\ \ \&\ \ (xfr\ (Entry,\ s_x)\ t_x)$

■

Figure D.9: Theorem 8.7 — One-to-one forward; proving existence and uniqueness of path

D.7 Proof of Pipelining Correctness (Theorem D.1)

Lemma D.8: *Unfold ReqN*
$\vdash$ $ReqN = req\ (Exit,\ Bot)$

Lemma D.9: *Unfold AccN*
$\vdash$ $AccN \ = \ acc\ (Exit,\ Bot)$

Lemma D.10: *Unfold ReqP*
$\vdash$ $ReqP = req\ (Top,\ Entry)$

Lemma D.11: *Unfold AccP*
$\vdash$ $AccP \ = \ acc\ (Top,\ Entry)$

Definition D.1: *Combinational representation is correct (Copy of Definition 7.15)*

$$
\begin{array}{l}
CombOk\ Pipe\ Comb \mathrel{\hat{=}} \\
\quad local\ open\ Pipe\ in \\
\quad\quad \forall\ T_i,\ T_o\ .\ Comb\ T_i\ T_o \iff travComb\ (Entry,\ T_i)\ (Exit,\ (T_o,\ Bot)) \\
\quad end
\end{array}
$$

Specification D.1: *Ordering (Copy of Specification 7.18)*

$$
\begin{array}{l}
\forall I,\ t_0,\ O,\ t_n\ . \\
\quad Match\ I\ O\ t_0\ t_n \iff \\
\quad\quad xfr\ (Top,\ Entry)\ t_0\ \& \\
\quad\quad travPipe\ (Entry,\ t_0)\ (Exit,\ t_n)\ \& \\
\quad\quad xfr\ (Exit,\ Bot)\ t_n
\end{array}
$$

Theorem D.1: *Pipelining correctness of framework*

$\vdash \forall Pipe,\ Match,\ Constraints$
$\quad (CoresOk\ (Pipe,\ Match,\ Constraints))\ \&$
$\quad (ConstraintsOk\ Pipe\ Constraints)\ \&$
$\quad (CombOk\ Pipe\ Comb)\ \&$
$\quad (FworkParamsOk\ (Pipe,\ Match,\ Constraints)\ Comb) \implies$
$\qquad \forall I,\ O\ .\ (Pipe\ I\ O)\ \&\ (envOk\ I\ O)\ \&\ (Constraints\ I\ O) \implies$
$\qquad\quad PipeliningOk\ (Pipe,\ Match,\ Constraints)\ I\ O$

- $CoresOk\ Pipe$
- $ConstraintsOk\ Pipe\ Constraints$
- $CombOk\ Pipe\ Comb$
- $FworkParamsOk\ (Pipe,\ Match,\ Constaints)$
- $(envOk\ I\ O)\ \&\ (Pipe\ I\ O)\ \&\ (Constraints\ I\ O)$

$\vdash$ $PipeliningOk\ Match\ I\ O$

BY `Unfold` *PipeliningOk* (* Definition 7.8 *)

(D.11) $\vdash$ $canEnter\ I\ O$

(D.12) $\vdash$ $isOneToOne\ Match\ I\ O$

Figure D.10: Theorem D.1 — Pipelining proof; first step

$\vdash$ *canEnter I O*

BY `Unfold` *canEnter* `(*` Definition 7.9 `*)`

$\vdash$ *sustain ReqN AccN*

$\vdash \forall\ t_r\ .\ \exists\ t_a\ .\ \mathit{holdCond}\ \mathit{ReqP}\ t_r\ \mathit{AccP}\ t_a$

BY `Lemmas D.8, D.9, D.10, D.11`
`(*` Unfold *ReqN*, *AccN*, *ReqP*, *AccP* `*)`

$\vdash$ *sustain* (*req* (*Exit, Bot*)) (*acc* (*Exit, Bot*))

BY `Lemma 8.1 with` $s_0 = \mathit{Exit}$, $s_1 = \mathit{Bot}$

■

$\vdash \forall t_r\ .\ \exists t_a\ .$
$\mathit{holdCond}\ (\mathit{req}\ (\mathit{Top},\ \mathit{Entry}))\ t_r\ (\mathit{acc}\ (\mathit{Top},\ \mathit{Entry}))\ t_a$

BY `Theorem 8.4 with` $s_0 = \mathit{Top}$, $s_1 = \mathit{Entry}$

■

Figure D.11: Theorem D.1 — Pipelining proof; transactions can enter the pipeline

$\vdash$ *isOneToOne Match I O*

BY `Unfold` *isOneToOne* (* Definition 7.10 *)

$\vdash \forall t_i \,.\, \mathit{XfrIn}\ t_i \implies \exists! t_o \,.\, \mathit{Match}\ I\ O\ t_i\ t_o$

BY `Theorem 8.7`

■

$\vdash \forall t_o \,.\, \mathit{XfrOut}\ t_o \implies \exists! t_i \,.\, \mathit{Match}\ I\ O\ t_i\ t_o$

BY `Theorem 8.8`

■

$\vdash \forall t_i, t_o \,.\, \mathit{Match}\ I\ O\ t_i\ t_o \implies (\mathit{XfrIn}\ t_i)\ \&\ (\mathit{XfrOut}\ t_o)$

BY `Specification 7.18`

■

Figure D.12: Theorem D.1 — Pipelining proof; one-to-one mapping between inputs and outputs

D.8 Proof of Functionality Correctness (Theorem 8.10)

- $Match\ I\ O\ t_0\ t_n$

$\vdash$ $Comb\ (getTransComb\ (I\ t_0))\ (getTransComb\ (O\ t_n))$

BY `Unfold Definition D.1 (*` $CombOk$ `*)`
`THEN Specification D.1`

- $xfr\ (Top,\ Entry)\ t_0$
- $travPipe\ (Entry,\ t_0)\ (Exit,\ t_n)$
- $xfr\ (Exit,\ Bot)\ t_n$

$\vdash$ $travComb\ (Entry,\ (getTransComb\ (I\ t_0)))$
$(Exit,\ (getTransComb\ (O\ t_n)),\ Bot))$

BY `Lemma 8.28 (*` $(...(I\ t_0)) = (transP\ Entry\ t_0)$ `*)`
`THEN Lemma 8.29 (*` $(...(O\ t_n)) = (transN\ Exit\ t_n)$ `*)`

- $travPipe\ (Entry,\ t_0)\ (Exit,\ t_n)$
- $xfr\ (Exit,\ Bot)\ t_n$

(D.14) $\vdash$ $travComb\ (Entry,\ transP\ Entry\ t_0)\ (Exit,\ (transN\ Exit\ t_n,\ Bot))$

Figure D.13: Theorem 8.10 — functionality proof; first step

- $travPipe\ (Entry,\ t_0)\ (Exit,\ t_n)$
- $xfr\ (Exit,\ Bot)\ t_n$

$\vdash$ $travComb\ (Entry,\ transP\ Entry\ t_0)\ (Exit,\ (transN\ Exit\ t_n,\ Bot))$

BY `Lemma 8.7` (* Induction on *travPipe* *)
`with` $\mathcal{P}\ (s_0,\ t_0)\ (s_n,\ t_n) =$
$travComb\ (s_0,\ transP\ s_0\ t_0)\ (s_n,\ (transN\ s_n\ t_n,\ Bot))$
`THEN Unfold Definition 7.18` (* *travComb* *)

- $match\ Entry\ t_0\ t_n$
- $xfr\ (Exit,\ Bot)\ t_n$

$\vdash$ $segComb\ Entry\ (transP\ Entry\ t_0)$
$(transN\ Entry\ t_n,\ Bot)$

BY `Lemma 8.30` (* $match \implies segComb$ *)

- $segComb\ Entry\ (transP\ Entry\ t_0)$
$(transN\ Entry\ t_n,\ Bot))$

$\vdash$ $segComb\ Entry\ (transP\ Entry\ t_0)$
$(transN\ Entry\ t_n,\ Bot))$

■

- $match\ Entry\ t_0\ t_x$
- $xfr\ (Entry,\ s_x)\ t_x$
- $travComb\ (s_x,\ (transP\ s_x\ t_x))$
$(Exit,\ (transN\ Exit\ t_n,\ Bot))$

(D.15) $\vdash$ $\exists s_x,\ T_x\ .$
$segComb\ Entry\ (transP\ Entry\ t_0,\ (T_x,\ s_x))\ \&$
$travComb\ (s_x,\ T_x)\ (Exit,\ (transN\ Exit\ t_n,\ Bot))$

Figure D.14: Theorem 8.10 — Functionality proof; induction on *travPipe*

- $match\ Entry\ t_0\ t_x$
- $xfr\ (Entry,\ s_x)\ t_x$
- $travComb\ (s_x,\ transP\ s_x\ t_x)\ (Exit,\ (transN\ Exit\ t_n,\ Bot))$

$\vdash\ \exists s_x,\ T_x\ .$
$segComb\ Entry\ (transP\ Entry\ t_0,\ (s_x,\ T_x))$ &
$travComb\ (s_x,\ T_x)\ (transN\ Exit\ t_x,\ Exit)$

BY `Instantiate concl with` $s_x{=}s_x$`,` $T_x{=}transN\ Entry\ t_x$

- $match\ Entry\ t_0\ t_x$
- $xfr\ (Entry,\ s_x)\ t_x$

$\vdash\ segComb\ Entry\ (transP\ Entry\ t_0)\ (transN\ Entry\ t_x,\ s_x))$

BY `Lemma 8.30 (*` $match \implies segComb$ `*)`

- $segComb\ Entry\ (transP\ Entry\ t_0)\ (transN\ Entry\ t_x,\ s_x)$

$\vdash\ segComb\ Entry\ (transP\ Entry\ t_0)\ (transN\ Entry\ t_x,\ s_x)$

■

- $travComb\ (s_x,\ transP\ s_x\ t_x)\ (Exit,\ (transN\ Exit\ t_n,\ Bot)$
- $xfr\ (Entry,\ s_x)\ t_x$

$\vdash\ travComb\ (s_x,\ transN\ Entry\ t_x)$
$(Exit,\ (transN\ Exit\ t_n,\ Bot))$

BY `Lemma 8.33`
`(*` $xfr\ (Entry,\ s_x)\ t_x \implies$ `*)`
`(*` $(transP\ s_x\ t_x\ =\ transN\ Entry\ t_x)$ `*)`

- $travComb\ (s_x,\ transP\ s_x\ t_x)\ (Exit,\ (transN\ Exit\ t_o,\ Bot))$

$\vdash\ travComb\ (s_x,\ transP\ s_x\ t_x)\ (Exit,\ (transN\ Exit\ t_o,\ Bot))$

■

Figure D.15: Theorem 8.10 — Functionality theorem; inductive case