Structuring instruction-sets with higher-order functions

John Byron Cook
B.Sci., The Evergreen State College, 1995

A dissertation submitted to the faculty of the
OGI School of Science & Engineering
at Oregon Health & Science University
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science & Engineering

January 2005

The dissertation "Structuring instruction-sets with higher-order functions" by John Byron Cook has been examined and approved by the following Examination Committee:

John Launchbury
Professor
Thesis Research Adviser

Mark Aagaard
Associate Professor, University of Waterloo

Todd Austin
Associate Professor, University of Michigan

Dan Hammerstrom
Professor

Richard Kieburtz
Professor Emeritus

Acknowledgements

I would like to thank my wife, children, and parents for their support during my Ph.D. research. Each of them have made numerous personal sacrifices that have given me this opportunity. Many friends have also generously helped along the way, offering childcare and other favors during my studies.

My advisor, John Launchbury, provided years of calm and patient guidance. Thank you. I also wish to thank Tom Ball, Per Bjesse, Nancy Day, Robert Jones, John Harrison, Sava Krstić, John Matthews, and Sriram Rajamani, Mary Sheeran, Don Syme, and the members of my committee for their technical comments and suggestions.

I would like to thank Intel's Strategic CAD laboratories. I learned a great deal about both formal verification and microprocessors while visiting Intel for a 6 month internship. Intel also provided financial support to my advisor which was used to pay for some of my expenses.

Prover Technology paid my salary and expenses during some of the time in which this research was done. Thank you.

I would also like to thank Marisa Anderson and Michelle Burbidge for all of their work: they typed much of the text in this dissertation from my handwritten notebooks. Finally, I wish to thank Jeff Henry who printed and submitted the final copies of this dissertation.

Contents

List of Figures

Abstract

Structuring instruction-sets with higher-order functions

John Byron Cook

Ph.D., OGI School of Science & Engineering

at Oregon Health & Science University

January 2005

Thesis Advisor: Dr. John Launchbury

In an effort to improve microprocessor performance, each generation of a microprocessor family's instruction-set architecture is typically extended with new features. For example, many modern microprocessors now support parallelism annotations, predication, speculative memory access, and SIMD-based multimedia instructions. These extensions allow a compiler or programmer to directly express instruction-level parallelism that is difficult for the microprocessor to find alone.

This dissertation focuses on the modeling and formal verification of microprocessor designs with instruction-set extensions. Inspired by ARM and IA-64, we develop several elementary instruction-set architectures that employ extensions. We also construct microarchitectural implementation of the instruction sets.

The specification and microarchitectural model in this dissertation are represented in a novel way: as the composition of functions between transition systems. We call these functions *transition system transformers*. In isolation, transformers can be used to model instruction-set extensions. Together, they can be used to model an entire machine.

This dissertation demonstrates that the extra structure available in transformer-based specifications and models can be used to help decompose a proof that the model implements a specification. We develop several proof strategies that make use of the transformer structure in this way. The contribution of this dissertation is the modeling and verification method that facilitates the decomposition of microarchitectural correctness proofs using instruction-set extensions.

Chapter 1

Introduction

Formal verification is the discipline of proving, with mathematics and logic, that a formal representation of a design is correct with respect to a specification. *Formal microarchitecture verification*, or *formal processor verification*, is the application of formal verification to a microarchitectural design. In recent years, a number of researchers have focused their attention on the verification of out-of-order microarchitectures, developing special techniques that use the structure inherent in these designs. This research has been fruitful—numerous papers have reported on the formal verification of relatively sophisticated microarchitectural models—and the proof techniques developed in this research are now being applied in industry.

However, while the research on formal verification has been directed at out-of-order machines, microprocessor designers have been experimenting with techniques that go beyond out-of-order execution. Architects are now adding constructs to the instruction-sets of their microprocessors that allow the programmer or compiler to explicitly declare optimizations. For example, a programmer can use VLIW-style *parallelism annotations* [22, 48, 61, 62] to specify a set of instructions that can be safely executed in parallel. A compiler can implement conditional codes with *predication* [5, 38] rather than branch instructions. A programmer can use a *speculative load instruction* [22, 39] rather than wait for a traditional load instruction to complete. When writing multimedia algorithms, a programmer can use SIMD-based *multimedia extensions* [14, 27, 28, 59].

These instruction-set extensions can potentially affect the way in which microprocessors are formally verified. For example:

- Instruction-set extensions provide a larger interface to the microprocessor, and therefore there are more ways in which instructions can interact within the microprocessor. This makes some verification techniques less tractable, particularly if they are based on executing the microarchitectural model on input vectors.

- Instruction-set extensions provide structure that can potentially be exploited during the correctness proof. That is, although the instruction-set is more complicated, it is closer to the microarchitectural design than has traditionally been true.

In this dissertation we pursue the second point. We introduce a method of modeling instruction-set extensions that facilitates their modular design. This method is based on the use of higher-order functions. We then develop several strategies for proving the correctness of a microprocessor model that implements instruction-set extensions. The first is a simple decomposition rule that can be used to break a proof obligation down into several proof obligations. The second rule can be used to simplify the overall proof by eliminating an obligation. We demonstrate, by example, how these two strategies can be used to decompose and simplify a microarchitectural correctness proof. The focus of this dissertation is on the development of the theory, and the application of the theory to decomposition. Therefore, rather than focusing on the proof of the example microarchitecture, on several occasions we either provide a proof outline or refer the reader to an algorithmic technique which has been successfully applied in the literature to prove a similar property.

The thesis presented in this dissertation is that higher-order functions facilitate both the design of architectural extensions and the proofs of their correctness. The key contributions of this dissertation are:

- a modeling method based on higher-order functions for extended instruction-sets and their microarchitectural designs;

- a decomposition proof strategy that leverages the proposed modeling method and

- a simplification strategy which can be applied to proof obligations left by the decomposition strategy. This is the result of an application of the theory of Parametricity [54, 66].

It should be noted that we will be making an important distinction not usually made in the literature on formal microarchitectural verification. We will use two languages when describing models and performing formal reasoning. The first language is a mathematical notation with a higher-order logic flavor. The second language is a restricted polymorphic functional programming notation. The functional notation's semantics are quite basic and support only bounded recursion with a set theoretic semantics. The literature typically blurs the distinction between the design language and the language in which reasoning is performed. The advantage to making this distinction will be made clear later.

1.1 Dissertation Synopsis

The remainder of this dissertation is organized as follows:

Chapter 2: Instruction-set extensions

In Chapter 2 we describe the microarchitectural concepts behind out-of-order and VLIW microprocessors and contrast them with the ideas underlying some common instruction-set extensions. We introduce two extended instruction-sets, which we call the *Washington Architecture* (WA) and the *Oregon Architecture* (OA). These architectures exhibit several of the common instruction-set extensions.

Chapter 3: Formal microarchitecture specification and verification

We survey the existing research on the formal modeling of microprocessors, their correctness criteria, and the techniques used to prove their correctness. The definitions and techniques developed in this chapter become the foundation from which we construct our new modeling and proof techniques.

Chapter 4: Modeling with transformers

We introduce a modeling technique based on the composition of a class of functions we call *transition system transformers.* Using these transformers, we develop a formal specification of the Oregon and Washington architectures, and microarchitectural designs that implement them. These microarchitectural designs draw influence from the 21264 [25], StrongARM [37], and the Itanium [35, 58, 18, 22, 26, 29, 47, 61].

Chapter 5: Proof with transformers

Chapter 5 develops several proof strategies that can be applied to correctness proofs of specifications and models that have been represented using transition system transformers. That is, if both the specification and the model are formed as the composition of transformers, this chapter provides applicable proof rules.

Chapter 6: Applying the theory of transformers

We apply the techniques from Chapters 3 and 5 to a decomposition of a proof that a microarchitectural design from Chapter 4 is correct.

Chapter 7: Conclusion

In the final chapter we conclude with a discussion of the strengths and weaknesses of the approach advocated in this dissertation, and discuss several leads for future research.

Chapter 2

Instruction-set extensions

In this chapter we introduce a number of concepts from out-of-order and VLIW microprocessor design. We then develop a simple predicated RISC instruction set called the *Washington Architecture* (WA) and an extended second-generation VLIW instruction-set, called the *Oregon Architecture* (OA) and explain how microarchitectural implementations of the instruction-sets might relate to their out-of-order and VLIW counterparts. OA and WA are used as examples throughout the dissertation.

2.1 Microarchitectures and microprocessors

The programmer's view of a microprocessor is typically a simple one. The programmer sees only the machine's visible state (the register-file, memory, etc.) and how each instruction effects that state. The programmer's view is often called the *instruction-set architecture* (ISA) and is typically represented as the conceptually simplest machine possible that implements the intention of the designer. The states that are reachable by the ISA are generally called the *ISA states.*

The microarchitect's perspective of a microprocessor, in contrast, is much richer. Performance goals force the microarchitect to design with techniques such as pipelining, buffering, speculative execution, and out-of-order and superscalar execution. We refer to the microarchitectural design as the *implementation*; and the reachable states of the microarchitectural model the *implementation states.* From this point forward, we will use the term *out-of-order* to refer to designs that mix techniques such as superscalar, out-of-order and speculative execution.

The tricks used in these out-of-order designs typically include techniques such as fetching and executing multiple instructions per cycle, issuing instructions based on resources rather than their position within a program, and the removal of false register dependencies with on-the-fly register renaming.

The basis for many of these high-performance techniques is the existence of parallelism between machine instructions. As an illustration of this parallelism, consider the program fragment in Figure 2.1 which implements the factorial function in a stylized RISC instruction-set. If we assume that the "branch if not equal to zero" instruction (`beqz`) in the figure is not taken for some time, (this is the sort of assumption that a microprocessor's branch predictor will often make) we can unfold the loop formed from instructions `103` to `106` and construct a data dependency graph like the one found in Figure 2.2. In this graph, an arrow is drawn between two instructions if they form a read-after-write dependency. If there is no path between two instructions, they are not dependent. Notice also that in the dependency graph we have labeled each instruction in the unfolding, beginning with `i0`. Every instruction is labeled in the order that it should be fetched.

High-performance microprocessors often maintain graphs like these in hardware. Using the analysis exhibited in this type of graph, a microprocessor might determine that it is possible to issue instruction `i4` into to an execution unit before instruction `i3`. In other words, the instructions can be executed out of program order; which is called *out-of-order execution.* The microprocessor might also choose to evaluate them in parallel (called *superscalar execution*). Because the value being placed into `r2` by instruction `i2` is not the same as the value of `r2` from instruction `i5`, the microprocessor could also replace the references to these registers with references to the labels of instructions which compute their values (see Figure 2.3). This is called *register renaming.*

The design of out-of-order microarchitectures is further complicated by the possibility of internal exceptions and external interrupts. For example, if an out-of-order microprocessor executes instruction `i1` before instruction `i0`, and the execution of instruction `i0` raises an internal exception, the processor could potentially be in a state where the register `r1` is set to `1` and the exception has been raised—which is a state that is not reachable by the ISA. To avoid these situations, high-performance microprocessors typically maintain

```
101: r2 ← load 400
102: r1 ← 1
103: beqz r2 107
104: r2 ← r2 - 1
105: r1 ← r1 * r2
106: jump 103
107: store 401 r1
```

Figure 2.1: Factorial function in a RISC instruction-set

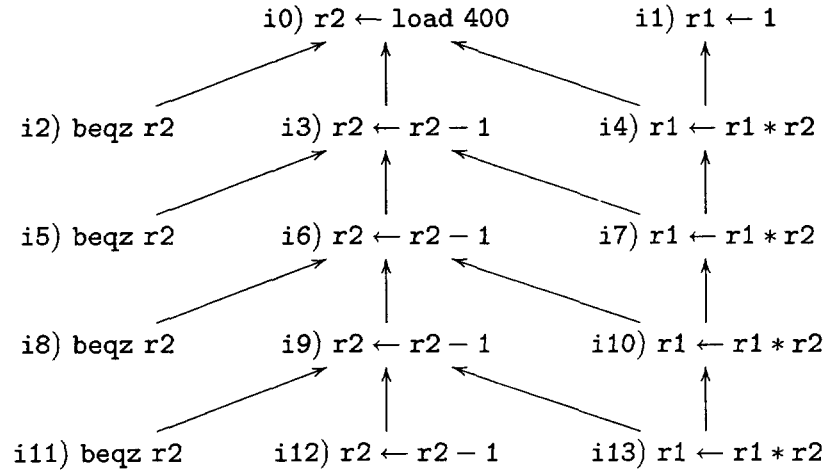


Figure 2.2: Data dependency graph of the factorial function in Figure 2.1

enough information so that they can place themselves back into a reachable ISA state in the event of an exception or interrupt.

2.1.1 An example of out-of-order design: the Intel P6 microarchitecture

Intel's P6 microarchitecture [24]—which underlies the Pentium Pro, Pentium II and Pentium III microprocessors—implements out-of-order execution using the following microarchitectural components: a *reorder buffer* maintains a finite region of the program's dependency graph; the *reservation stations* maintain the portion of the reorder buffer that remains to be computed; the *execution units* compute, in parallel, the destination values

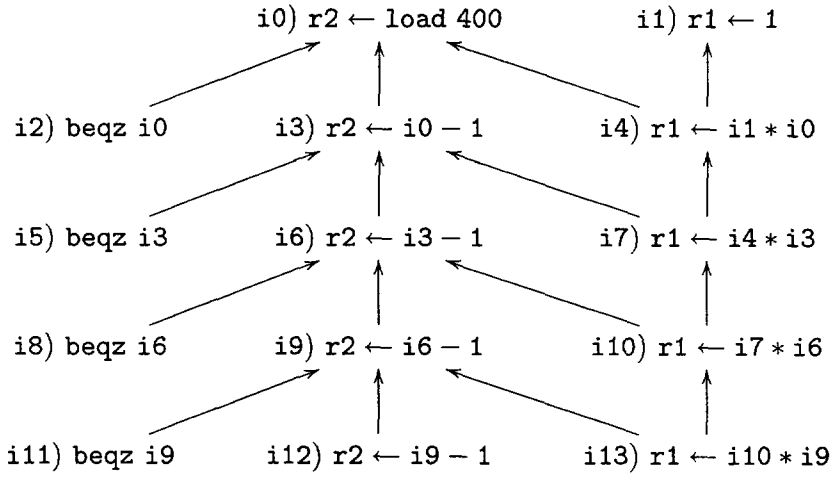
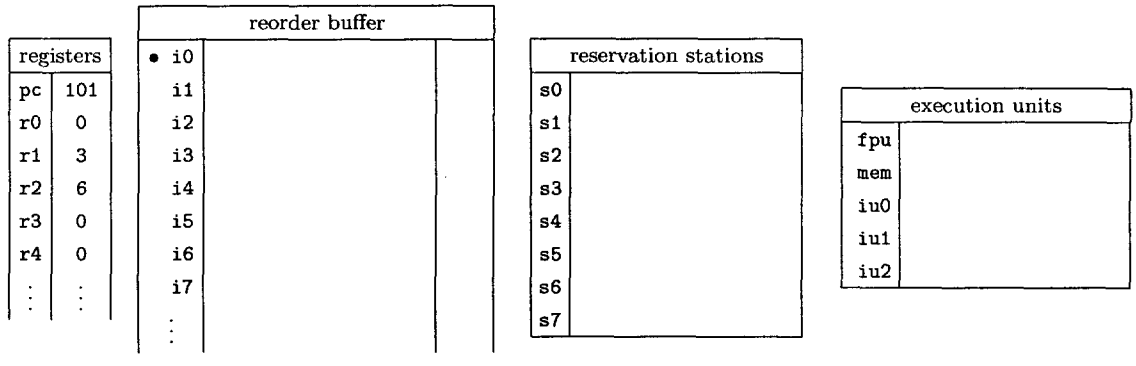


Figure 2.3: Data dependency graph from Figure 2.2 with source references renamed

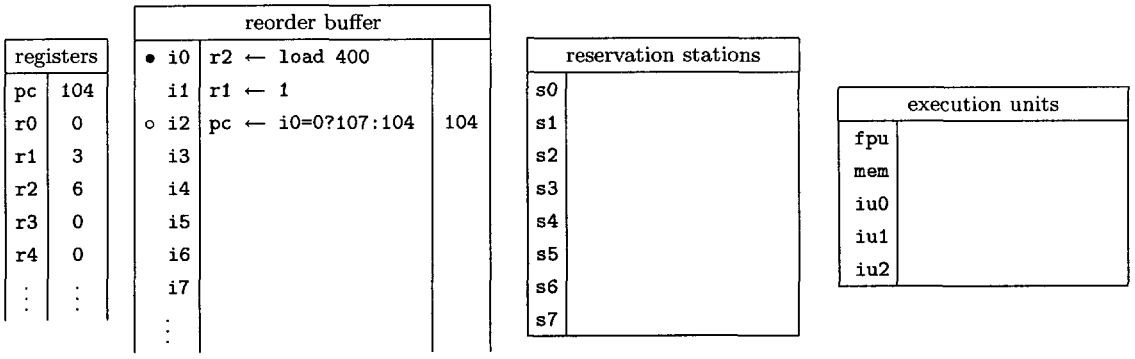
for the instructions in the reservation stations; and the *register-file* represents the current ISA state.

To demonstrate how the P6 works, we can partially execute the encoding of the factorial function from Figure 2.1 in a mock P6-like machine. We begin with an empty reorder buffer, empty reservation stations, and a program counter (pc) pointing to instruction 101:



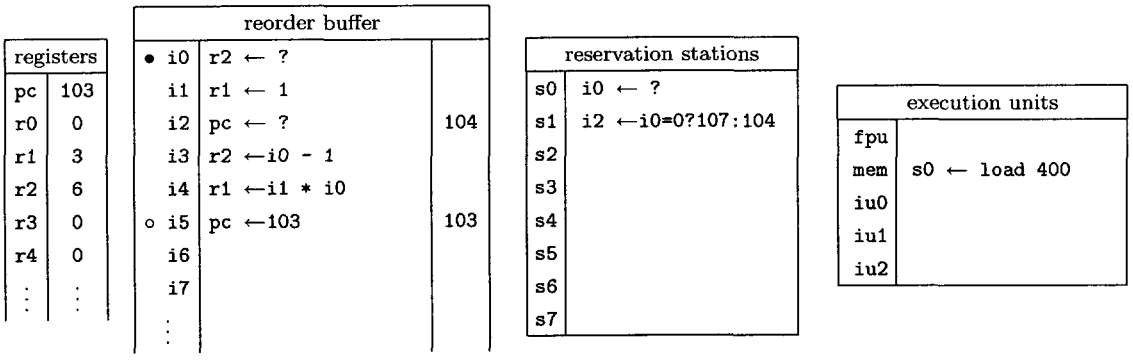
We will assume that the mock machine can fetch up to three instructions per cycle. Therefore, on the first cycle, we insert instructions 101, 102 and 103 into the machine. The internal representation of the beqz instruction is pc ← r2=0?107:104 , which should be interpreted as "if r2 equals 0 then set the program counter to 107, otherwise set it to 104." In the next state below, notice how the reorder buffer constructs the active region of

the data dependency graph: as instructions are loaded into the buffer, register references are replaced with pointers into the graph. In this case, `r2` is replaced with `i0`. Note that the reorder buffer locations are the same as the instruction labels in Figure 2.3.



The reorder buffer is designed as a circular queue: the symbol • is used to represent the front of queue, and o represents the end. The third column in the reorder buffer is used to store branch predictions. In the above state, the machine has guessed that the result of `i2` will eventually resolve to `104`. When retiring instruction `i2`, if the machine discovers a misprediction, it will bring itself into a corrected state by clearing the reorder buffer, registration stations, and execution units; and saving the value of `i2` to location `pc` in the register-file.

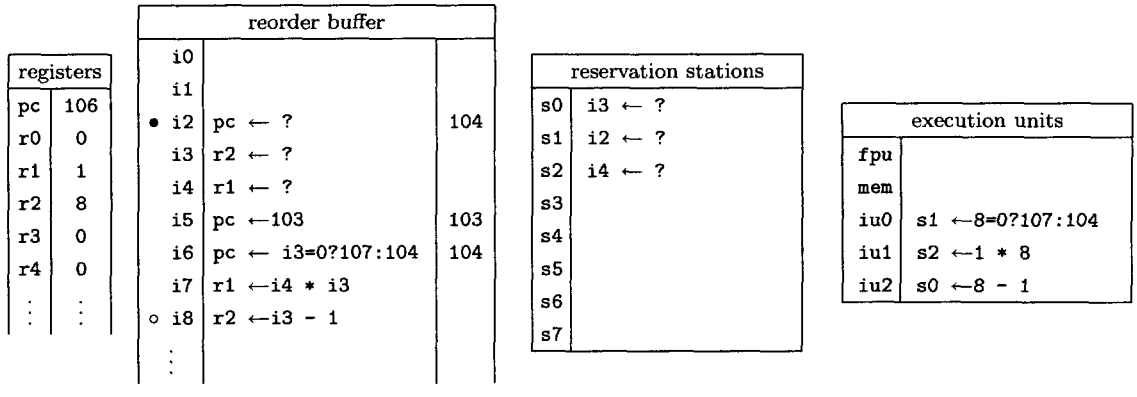
On the next cycle, we can simultaneously load instructions `104`, `105` and `106` and begin to execute instructions `i0` and `i1`:



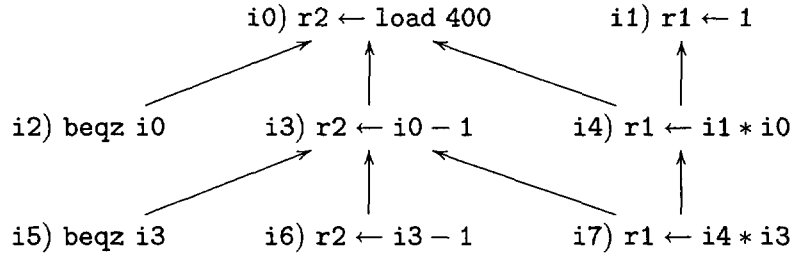
The P6 implements *in-order writeback*, meaning that the results of instructions are written back to the register file in program order. This policy ensures that the register file is always in an ISA state, maintaining the programmer's illusion of sequential execution.

For example, in the state above the execution of instruction `i1` is complete (the value it is computing for its destination register is known). However, we cannot yet safely write its value to the register-file until we know that executing instruction `i0` has not caused an exception.

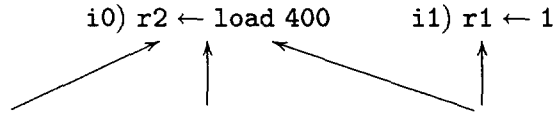
We will assume, on the next cycle, that the load instruction is complete. We can therefore write the values of instructions `i0` and `i1` to the register file and remove them from the reorder buffer. With the value of `i0` known, we can also issue instructions `i3`, `i4`, and `i5` into the integer execution units:



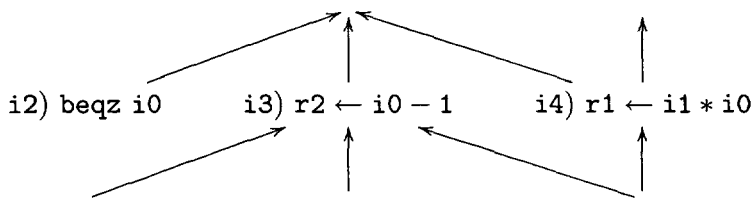
This demonstration highlights how the P6 finds parallelism between instructions in a sequential program. For example, it found the parallelism between instructions `i0` and `i1`, and between `i2`, `i3`, and `i4`. Recall the program dependency graph in Figure 2.3. In the current state, the machine has traversed halfway down the graph:



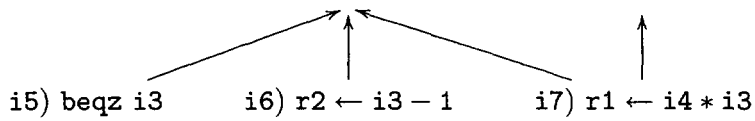
The machine has computed and retired the first row of the graph:



it is currently executing the second row of the graph:



and it has fetched the third row of the graph:



2.2 Techniques for making a microprocessor's data dependency graph explicit

Unfortunately, P6-style connected graph structures can lead to large and complex microprocessors. Graph based machines, such as the P6, are difficult to design and debug, and typically have long critical paths, which inhibit faster clock speeds. In response to this, microprocessor companies have, over the years, experimented with VLIW machines, which do not suffer these weaknesses.

Like the P6, VLIW machines typically fetch multiple instructions on each cycle. These groups of instructions are sometimes called *packets*. Unlike the P6, VLIW machines do not provide reorder buffers or reservation stations. Instead, their hardware is typically a simple vector of execution units with a register file. The programmer or compiler must then know the latencies of the execution units of the particular machine and schedule the instructions appropriately.

For example, assume that memory instructions have a one cycle latency, multiplications have a three cycle latency, and that branches have no latency. Also, assume each packet has the following format:

```
fp ; mem ; int0 ; int1 ; int2
```

```
101: nop ; r2 ← load 400 ; r1 ← 1       ; nop           ; nop
102: nop ; nop           ; nop          ; nop           ; nop
103: nop ; nop           ; beqz r2 105  ; nop           ; nop
104: nop ; nop           ; r2 ← r2 - 1  ; r1 ← r1 * r2  ; jump 103
105: nop ; store 401 r1  ; nop          ; nop           ; nop
```

Figure 2.4: VLIW factorial function

where `fp` is a floating point instruction, `mem` is a memory operation and `int0` through `int2` are integer instructions. Figure 2.4 contains a VLIW source fragment for the factorial function. We use the `nop` instruction to separate dependent instructions sufficiently so that the latencies of the execution units do not cause incorrect results. A VLIW compiler or programmer must perform considerable analysis to build this program fragment, but the hardware that executes it is simple.

For example, we can run a VLIW machine on the program fragment in Figure 2.4. We begin execution with an empty machine:

registers	
pc	101
r0	0
r1	0
r2	0
r3	0
r4	0
⋮	⋮

execution units	
fpu	
mem	
iu0	
iu1	
iu2	

On the first cycle we load and execute the instructions from memory location `101`. This results in a state in which the register `pc` has been incremented, register `r1` has been set to 1, and the load instruction has been issued to the memory unit:

registers	
pc	102
r0	0
r1	1
r2	0
r3	0
r4	0
⋮	⋮

execution units	
fpu	
mem	r2 ← load 400
iu0	
iu1	
iu2	

In the next cycle we load the `nop` instructions in memory location 102. We have placed these `nop`s into the program to provide time for the `load` instruction to calculate the value of `r2`.

registers	
pc	103
r0	0
r1	1
r2	0
r3	0
r4	0
⋮	⋮

execution units	
fpu	
mem	r2 ← load 400
iu0	
iu1	
iu2	

On the next cycle we fetch the branch instruction in location 103:

registers	
pc	104
r0	0
r1	1
r2	8
r3	0
r4	0
⋮	⋮

execution units	
fpu	
mem	
iu0	pc ← 8=0?103:107
iu1	
iu2	

We then fetch the parallel subtract, multiply, and jump instructions:

registers	
pc	103
r0	0
r1	1
r2	8
r3	0
r4	0
⋮	⋮

execution units	
fpu	
mem	
iu0	r2 ← 8 - 1
iu1	r1 ← 1 * 8
iu2	

Note that, in effect, a VLIW machine is like the back-end execution core of our mock P6-like machine. It would be expected that a compiler or the programmer had performed the analysis of the reorder buffer before executing the program.

The drawback to VLIW microprocessors is that, from one processor generation to the next, the latencies of instructions often change. This destroys binary-code compatibility and requires that programs be re-compiled. Another problem is that, when there is no parallelism to fill up the slots in a VLIW packet, the fetch bandwidth is wasted loading nops. Consequently, the success of VLIW in mainstream computing has been limited.

However, a trend in microprocessor design is to adapt the ideas from VLIW design in such a way that the compatibility and space problems are mitigated. The so-called *explicitly parallel* or *second generation VLIW* instruction-sets provide VLIW-like extensions to RISC. Rather than specifying exactly how the instructions should be executed in the machine, in some cases, these extensions allow the programmer to simply express the existence of parallelism, allowing the machine the option of implementing parallelism-based optimizations.

For example:

Parallelism annotations [61] declare which instructions within a program can be executed out-of-order or in parallel. The microprocessor then uses this information to schedule instructions into execution units. This feature appears in Intel's IA-64 [22], and Compaq's Araña [62].

Predication [5] expresses conditional execution using data dependence rather than branch instructions. IA-64 and ARM [38] are examples of predicated instruction sets.

Speculative instructions [22] behave like their traditional counterparts—however the exceptions they might cause are raised only when and if the data they compute is used. IA-64 and PA-RISC [39], for example, both support speculative loading mechanisms.

Multimedia instructions provide SIMD-based instruction-set extensions which are well-suited for multimedia computation. MMX [14], AltiVec [27], and 3DNow [28, 59] are examples of multimedia specific instruction-set extensions.

2.3 The Washington architecture

We now describe a simple example of a predicated instruction-set, which we call the *Washington Architecture* (WA). The extension to RISC that WA provides is the **if** syntax. The semantics of the following instruction is that it should be executed only in the case that the value of the predicate register `p1` is `true` in the predicate register-file:

$$r1 \leftarrow r1 * r2 \textbf{ if } p1$$

WA allows us to encode conditional execution without branches and jump instructions. For example, we re-encode the RISC code fragment:

```
        beqz r1 L1
        r2 ← r4 * r5
        jmp L2
L1:     r2 ← r4 + r5
L2:     nop
```

As this:

```
p1,p2 ← r1==0
r2 ← r4 * r5  if p1
r2 ← r4 + r5  if p2
```

Where the predicate `p1` will be set to `true` if `r1` equals zero and `p2` is assigned to the negation of `p1`. This is a more efficent encoding because the branch mechanism (and branch prediction unit) is not used. This feature is similar to the form of predication in the ARM and IA-64 instruction-sets.

2.4 The Oregon architecture

Next we develop a simple example of a second-generation VLIW instruction-set as an extension to WA, which we call the *Oregon Architecture* (OA). This instruction-set combines predication with explicit parallelism constructs. Figure 2.5 provides an example OA

encoding of the factorial function.

To see how these extensions fit into OA, look at Figure 2.5 which contains an OA encoding of the factorial function:

- As in VLIW, an OA program is a finite sequence of *packets*, where each packet consists of a fixed number of instructions. In this case, packets consist of three instructions. Programs are addressed at the packet-level. That is, instructions are fetched in packets, and branches can jump only to the beginning of a packet.

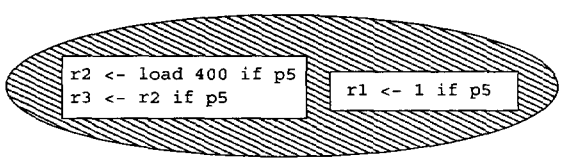
- Instructions are annotated with *thread identifiers*. Instructions with equal identifiers should be executed sequentially. Instructions with independent identifiers can be executed in any order. For example, the `0` in the `load` instruction declares that instructions with thread identifiers that are not equal to `0` can be executed in any order with respect to the `load` instruction.

- Packets can be annotated with the directive `FENCE`, which directs the machine to fully calculate all in-flight results before executing the following packet.

- As in WA, instructions in OA are predicated on Boolean-valued predicate registers. For example, the `load` instruction will only be executed if the value of `p5` is true in the current predicate register-file state.

One way to view the thread identifiers and fences in OA is with directed graphs whose nodes are the sets of threads that occur between fence directives. These sets are analagous to basic blocks, using compiler terminology. The idea is that an OA machine will execute one basic block at a time. In this manner, all values computed in previously executed basic blocks are available to all threads in the current basic block.

For example, the fence directive after packet `101` instructs the microprocessor to retire the active threads before executing the following packet. Assuming that packet `100` issues a fence directive, packet `101` forms its own basic block:

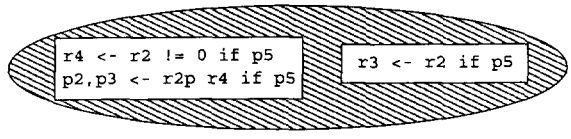


```
101:  r2 ← load 100 if p5 in 0
      r1 ← 1 if p5 in 1
      nop
FENCE

102:  r4 ← r2 != 0 if p5 in 0
      p2,p3 ← r2p r4 if p5 in 0
      r3 ← r2 if p5 in 1
FENCE

103:  r2 ← r2 - 1 if p2 in 1
      r1 ← r1 * r3 if p2 in 0
      pc ← 102 if p2 in 2


104:  store 401 r1 if p3 in 3
      pc ← 105 if p3 in 2
      nop
FENCE
```

Figure 2.5: Oregon Architecture (OA) factorial function

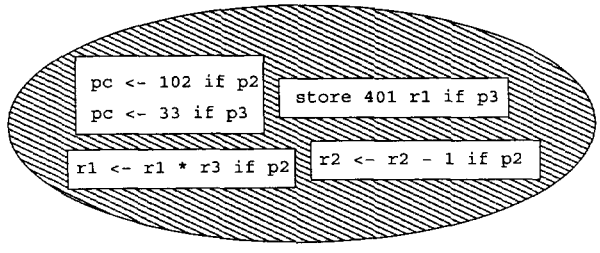
In this picture, the ovals are used to represent basic blocks, and boxes to represent threads. Instructions within a thread must be executed in order. Threads, however, can be executed in any interleaving-order with other threads. Because packet 101 is its own basic block, the machine is required to synchronize the state before executing the next packet.

Because packet 102 and 103 are separated by a fence, packet 102 forms its own basic block:

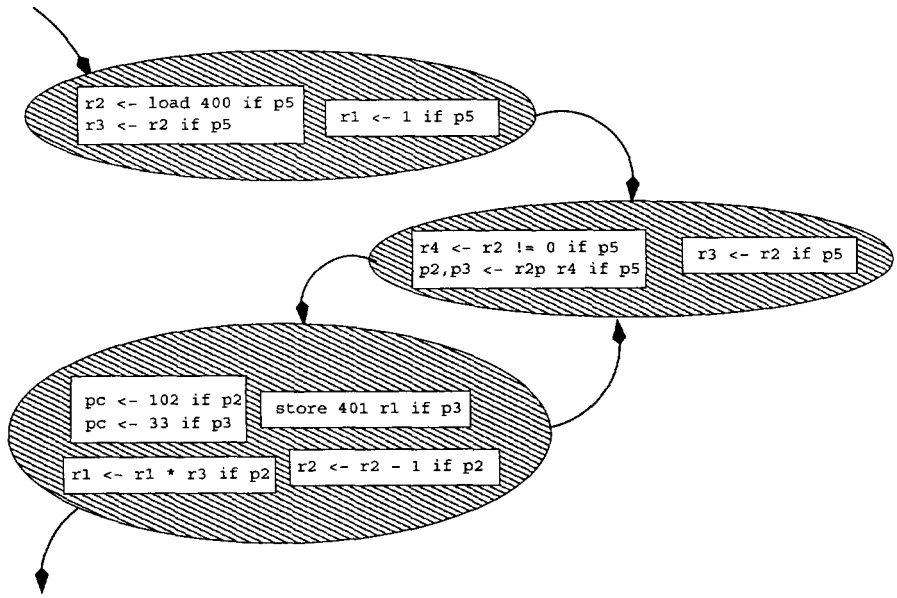


The comparison and copy instructions set the predicate register p2 to true if r2 is not equal to 0. The value of p3 is set to the negation of p2.

Because packet 103 is not fenced, but packet 104 is, the next basic block is formed from packets 103 and 104:



Assignments to the program counter within a basic block are visible to the machine's fetch mechanism only after a fence directive has been issued. That is, assignments to pc tell the machine where to fetch from after executing the next fence. Therefore, a trace of an OA program can be viewed as an infinite path through the finite directed graph formed by basic blocks and their successors:



Reasoning about these basic blocks is analogous to the sort of control calculation the P6 performs during execution [57]. For example:

- Figure 2.1 uses a conditional branch in the place of the predicate calculation. The P6 with branch speculation might predict that the branch is not taken and issue the multiplication and subtraction before calculating the condition. In this case the branch prediction mechanism is acting as a predicate register file.

  The OA program calculates a predicate, issues instructions from both sides of the potential branch, and only retires the instructions that satisfy the predicate.

- In Figure 2.1 much of the instruction-level parallelism discovered in Figure 2.2 is implicit. The P6 analyzes register references to find parallelism (e.g. between the subtraction and multiplication instructions).

  In OA, the compiler or programmer declares the dependencies between instructions. In addition, the final values in the registers may not correspond to any interleaving of the original instructions, due to latencies between storing and fetching values from memory and the register file.

One constraint that OA assumes the programmer to maintain is that no read-after-write or write-after-write hazards shall occur between instructions in different threads.

For example, the following code (which contains a read-after-write hazard) will not be allowed:

```
r1 ← r2 + r3 in 0
r4 ← r1 + r3 in 1
```

We assume that the compiler or programmer will not violate this rule.

As another example, this code (which contains a write-after-write hazard) will also not be allowed:

```
r1 ← r2 + r3 in 0
r1 ← r3 + r4 in 1
```

2.5 Summary

Modern instruction-set extensions allow the compiler or programmer to specify parallelism between instructions, a job that has been traditionally left to the microprocessor. However, unlike VLIW instruction-sets, they typically do not directly expose the structure of the microarchitecture and the latencies of its execution units.

In this chapter we have surveyed the landscape of microprocessor design and introduced WA and OA, elementary instruction-sets that illustrate several features that appear within the instruction-sets of some popular microprocessors: predication and concurrency annotations. In this chapter we saw that the particular combination of extensions in OA forms a language in which thread parallelism can be expressed within basic blocks. However, the semantics of each feature could be changed—which would result in a different sort of machine language. In Chapter 4 we explore a method of modeling that allows us to isolate the meaning of each extension.

Chapter 3

Formal microarchitecture specification and verification

In this chapter we describe the formalism of transition systems, which are commonly used in the literature for specifying and modeling microarchitectures. We then describe several correctness criteria on transition systems that are commonly proved of microarchitectural models. The material from this chapter forms the basis from which the theory and the example in later chapters are constructed.

3.1 Preliminaries

As mentioned in Chapter 1, we are distinguishing between programming language syntax and mathematical semantics. The distinction will be made explicitly with fonts. Typewriter font will denote syntax in a programming language. Mathematical font will indicate mathematical and logical expressions. The use of semantic brackets ($[\![]\!]$) will be used to indicate the mathematical *meaning* of programming language syntax. We will use sans serif font to name mathematical definitions.

For example, we provide the mathematical definition of the identity function, Id:

Definition 1 (Id) $(a, b) \in \mathsf{Id} \triangleq a = b$.

We also provide an analogous definition in our programming language syntax:

Definition 2 (`id`) *The function* `id` *with type* `a -> a` *is defined as* $[\![\texttt{id}]\!] \triangleq \mathsf{Id}$.

In this dissertation we assume that the reader is familiar with functional programming notation. We will, however, introduce several of the more obscure concepts that we are

borrowing from the functional programming language Haskell [36]: qualified types, and the do-notation. We will also introduce the concepts and code behind a microarchitectural modeling library written in this programming language notation.

Type-classes

Type-classes [40] facilitate overloading in a parametrically polymorphic typing system. Suppose, for example, that we want to write a function that takes a register-file and returns the value of the program counter in the register-file, with the constraint that this function should work for register-files over any register-type. As a first attempt, we might try to write a function with type:

```
type RF a b = a -> b
read_pc :: RF a b -> b
```

That is, the type `RF a b` is synonymous for functions from `a` to `b`. Also, for any type `a` and `b`, the function `read_pc` takes an `RF a b` and returns a `b`. Unfortunately, we cannot write the intended function with this type in our language. The problem is that the type `a` is a parameter of `RF a b`, i.e. it is a place holder for any type. The function `read_pc` should work identically for the integers, Booleans, strings, etc. The consequence of this is that it is impossible to know which element of type `a` is the program counter, or if one even exists. Type-classes address this problem. With type classes, we can constrain the function `read_pc` such that it will be applicable only to types that have an element called `pc`. First we define a type-class called `Register`:

```
class Register r where
    pc :: r
```

This declaration partially defines a set of types in which a predicate is true: all of the types in this set have an element named `pc`. The reader should read the declaration as follows: "if `r` belongs to the type-class `Register`, than it must have an element called `pc`". For each type in the class `Register`, we must declare that it belongs to the class, and point to an element that will represent `pc`. For example, we can declare 32-bit words to be in the `Register` type-class:

```
instance Register Word where
    pc = 0
```

That is, for the type `Word`, `pc` equals `0`. We can now write a function with the original intended behavior:

```
read_pc rf = rf pc
```

This function has the following qualified type:

```
read_pc ::  Register a => RF a b -> b
```

This type should be read as "if the type `a` is an instance of the type-class `Register`, then `read_pc` is a function from `RF a b` to `b`." In the definition of `read_pc`, when the function `rf` is applied to `pc`, the value of `pc` of type `a` is used. For example, if `wrf` has type `RF Word Bool`, and $[\![\texttt{wrf 0}]\!] = 5$ then

$$[\![\texttt{read\_pc wrf}]\!] = 5$$

As another example, imagine trying to write a polymorphic function, `elem`, that determines if an argument is an element of a list. To do this we require an equality function (==) that works for all types:

```
elem x []     = False
elem x (y:ys) = if x==y then True
                else elem x ys
```

The idea is that == should be defined on a number of types. We can make this assumption explicit by defining an equality type-class:

```
class Eq a where
    (==) :: a -> a -> Bool
```

This declaration states that == is a function that should be defined for all types that are elements of the `Eq` type-class. A function defined with == will have this type constraint in its type. So, for example, the type of `elem` is:

```
elem :: Eq a => a -> [a] -> Bool
```

We will use the type-class `Collect` as a common interface to set-like structures:

```
class Collect c where
        unit :: a -> c a
        map :: (a -> b) -> c a -> c b
        join :: c (c a) -> c a
        union :: c a -> c a -> c a
```

This definition states that if a type `c` is an element of `Collect`, then the functions `unit`, `map`, `join`, and `union` will be defined on `c`. The meaning of these functions depends on the interpretation assigned to them for each `c`.

Let us look at some possible implementations. We can define a basic set-like structure: `type One a = a`. That is, `One` is the type that represents singleton sets. `One` can be declared an instance of `Collect`:

```
instance Collect One where
     unit = id
     map f x = f x
     join = id
     union x y = x
```

In this case, the type of `join` is `One (One a) -> a`. Note that, because `Onea = a`, that `One (One a) -> a` is equivalent to `a -> a`.

We will assume that finite sets are provided by the language as a built-in construct called `FSets`. This too is an instance of `Collect`, where:

$$
\begin{aligned}
[\![\texttt{unit}]\!] &\triangleq \{\cdot\} \\
[\![\texttt{map}]\!] &\triangleq \lambda f.\ \lambda A.\ \{f\ a | a \in A\} \\
[\![\texttt{join}]\!] &\triangleq \bigcup \\
[\![\texttt{union}]\!] &\triangleq \cup
\end{aligned}
$$

Because `FSet` is built-in we define the instantiation with a direct semantic definition.

The do-notation

In some circumstances, we make use of a convenient set-comprehension like notation, called the *do-notation*. A do-notation expression like this:

```
data Opcode = ADD Reg Reg Reg    | ADDI Reg Reg Word | SUB Reg Reg Reg
            | SUBI Reg Reg Word | MLT Reg Reg Reg   | MLTI Reg Reg Word
            | DIV Reg Reg Reg   | DIVI Reg Reg Word | CNT Reg Word
            | BNEZ Reg Word     | BEQZ Reg Word     | NEQZ Reg Reg
            | EQZ Reg Reg       | NEG Reg Reg       | BUBBLE
```

Figure 3.1: `Opcode`, A RISC instruction-set type

```
do { x <- a; y <- b; z <- f y; unit (z,x) }
```

is translated accordingly:

```
do { x <- a
   ; y <- b
   ; z <- f y
   ; unit (z,x)
   }
```

$\rightsquigarrow$

```
let bind x f = join (map f x)
in     a `bind` (\x.
       b `bind` (\y.
   (f x) `bind` (\z.
        unit (z,x))))
```

In this code we see several new syntactic structures. The back-quotes around `bind` indicate that `bind` is being used in an infix position. The syntax $\backslash x.E$ allows us to introduce a function where x is the function's parameter and E is the function's body.

In the finite-set interpretation of `Collect` and $[\![\cdot]\!]$, the meaning of the translation of this do-notation code represents the following finite set:

$$\{(z,x) \mid x \in [\![\texttt{a}]\!] \wedge y \in [\![\texttt{b}]\!] \wedge z \in [\![\texttt{f}]\!](y)\}$$

In the `One` interpretation, the translation represents:

$$([\![\texttt{f}]\!]([\![\texttt{b}]\!]), [\![\texttt{a}]\!])$$

which equals $[\![\texttt{(f b,a)}]\!]$.

3.1.1 Transactions

A key concept used later in this chapter is the idea of *transactions* [2, 50]. A transaction is like a machine-level instruction, with data bundled in the operands. Figure 3.2 contains

```
type Trans = ( [(Reg,Maybe Int)] , Opcode , [(Maybe Reg,Maybe Int)])
```

Figure 3.2: Trans, the transaction type

```
make_trans (ADD x y z)
       = ((x,Nothing),ADD x y z,[(Just y,Nothing),(Just z,Nothing)])
make_trans (SUB x y z)
       = ((x,Nothing),SUB x y z,[(Just y,Nothing),(Just z,Nothing)])
```

etc...

Figure 3.3: The transaction combinator make_trans

```
read_stage :: RF -> Trans -> Trans
read_stage rf (dsts,opcode,srcs) = (dsts,opcode,srcs')
   where srcs' = map (rd rf) srcs
         rd rf (Just r,z) = (Just r,Just (readEnv rf r))
         rd rf x = x
```

Figure 3.4: The transaction combinator read_stage

```
alu_stage :: Trans -> Trans
alu_stage ( [(r1,_)], ADD, ((r2,Just x):(r3,Just y):s))
   =      ( [(r1,Just (x + y))], ADD, ((r2,Just x):(r3,Just y):s))
alu_stage ( [(r1,_)], SUB, ((r2,Just x):(r3,Just y):s))
   =      ( [(r1,Just (x - y))], SUB, ((r2,Just x):(r3,Just y):s))
```

etc...

Figure 3.5: The transaction combinator alu_stage

```
wb_stage :: Trans -> RF -> (RF,Trans)
wb_stage  (dsts,i,srcs) rf = (rf',(dsts,i,srcs))
   where rf' = foldl writeback rf dsts
         writeback rf (r,Just x) = updateEnv rf (r,x)
```

Figure 3.6: The transaction combinator wb_stage

```
bypass :: Trans -> Trans -> Trans
bypass (dsts1,i1,srcs1) (dsts2,i2,srcs2) = (dsts2,i2,srcs3)
  where
  srcs3 = map (bp dsts1) srcs2
  bp dsts (Just x,y) =
     let dom = map fst dsts
         func x = case filter (\(a,Just b) -> a==x) dsts of
                    (a,Just b):c -> Just b
     in if x `elem` dom then (Just x,func x) else (Just x,y)
  bp dsts x = x
```

Figure 3.7: The transaction combinator bypass

a definition of a type `Trans` which models the concept of a transaction. Within this type declaration, the types `Opcode` and `Maybe` are used. `Opcode` (see Figure 3.1) is used to denote a RISC instruction-set. The type `Maybe` is like the ML type `option`:

```
data Maybe a = Just a | Nothing
```

This means, for example, that a value of type `Maybe Int` is either a `Nothing`, or it is a `(Just n)` for some integer n.

We provide a number of functions to construct and inspect transactions. For example, `make_trans` (Figure 3.3) can be used to create a transaction from an instruction. If the incoming instruction is `ADD r1 r2 r3`, then `make_trans` will produce the transaction:

```
([(r1,Nothing)],ADD,[(Just r2,Nothing),(Just r3,Nothing)])
```

This transaction represents an instruction with no data available. That is: the value of `r1` is not available (`Nothing`), nor are the values of `r2` and `r3`.

The function `read_stage` (Figure 3.4) takes a register-file and a transaction and returns a new transaction in which the value of the source operands have been placed with the references. For example, if in a register-file `r2` equals 2 and `r3` equals 3, then `read_stage` would take the above transaction and produce:

```
([(r1,Nothing)],ADD,[(Just r2,Just 2),(Just r3,Just 3)])
```

This transaction represents the instruction with its source operand values known.

The function `alu_stage` (Figure 3.5) could be used to take this transaction, perform addition on the source operands, and place the computed value in the destination field:

```
([(r1,Just 5)],ADD,[(Just r2,Just 2),(Just r3,Just 3)])
```

This transaction represents the instruction where `r1` is assigned 5.

The function `wb_stage` (Figure 3.6) can be used to update a register file with the bindings from the destination operands `[(r1,Just 5)]`. In this definition we see an application of `foldl`, a commonly used function in functional programming that iteratively applies a function to a list.

The final transaction function is `bypass` (Figure 3.7), which can be used to mediate data dependencies between transactions. Suppose a transaction is on its way through a microprocessor's pipeline with an outdated `r1`-value in its source operands:

```
old = ([(r5,Nothing)],SUB,[(Just r1,Just 8),(Just r9,Just 2)])
```

Also, suppose that another transaction is traveling through the machine with a recently computed value for `r1`:

```
new = ([(r1,Just 5)],ADD,[(Just r2,Just 2),(Just r3,Just 3)])
```

In this case we could use `bypass` to construct a new transaction based on `old` with an updated `r1`-value:

```
bypass new old = ([(r5,Nothing)],SUB,[(Just r1,Just 5),(Just r9,Just 2)])
```

3.1.2 Connecting transaction combinators and hardware

The purpose of these transformers is to simplify the development of hardware. Therefore it is natural to wonder how we might map from the transaction combinators down to wires and registers. Rather than provide a rigorous and complete mapping we will take `alu_stage` as an an example combinator and demonstrate its compilation.

This unit takes in a set of wires representing the transaction and produces a set of outputs that represent the resulting transaction. Each type can be represented with a finite set of wires (given that the sizes of lists are bounded). For example, an output of type `Maybe Int` can be represented with 32 wires for `Int` and one additional wire for the `Just` and `Nothing` constructors. Tuples are implemented simply as the concatenation of types.

Each equation from Figure 3.5 can be implemented as a circuit, and the overall function can then be the composition of these circuits with a large mux that implements the pattern matching. That is: the hardware can speculatively compute the answer for each equation and then choose the right one with a mux by inspecting the input transaction with a pattern matching circuit. As pictured in Figure 3.8, if the pattern match in the `ADD` case succeeds, the output would be the same as the input except that the with the data line

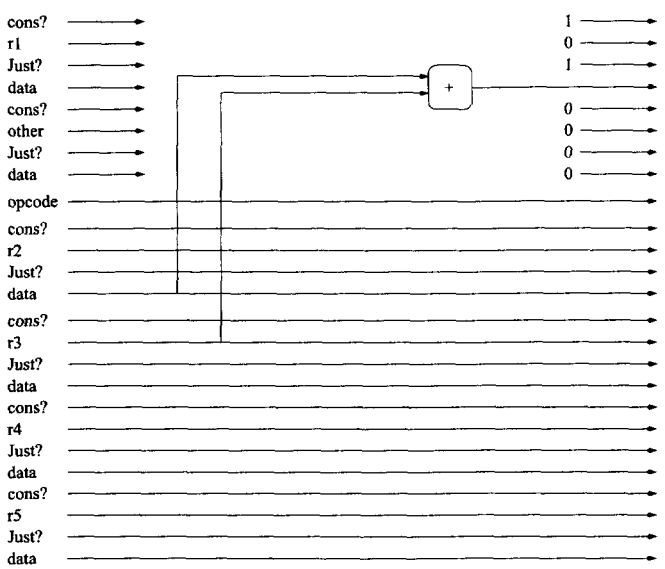


Figure 3.8: Schematic diagram for `alu_stage` in the `ADD` equation

for `r1` would be in with the data from `r2` added with `r3`. Figure 3.9 contains a circuit implementing the pattern match for the `ADD` case from Figure 3.5.

3.2 Specifying and modeling with transition systems

In microarchitecture verification, with few exceptions, both instruction-set specifications and microarchitectural implementations are represented as transition systems. Although the formalisms differ slightly from paper to paper, the following notation suffices for our discussion.

A transition system is a structure with three elements:

- A set of initial states,
- A next state relation, and
- A function that labels or projects out the visible parts of system's states.

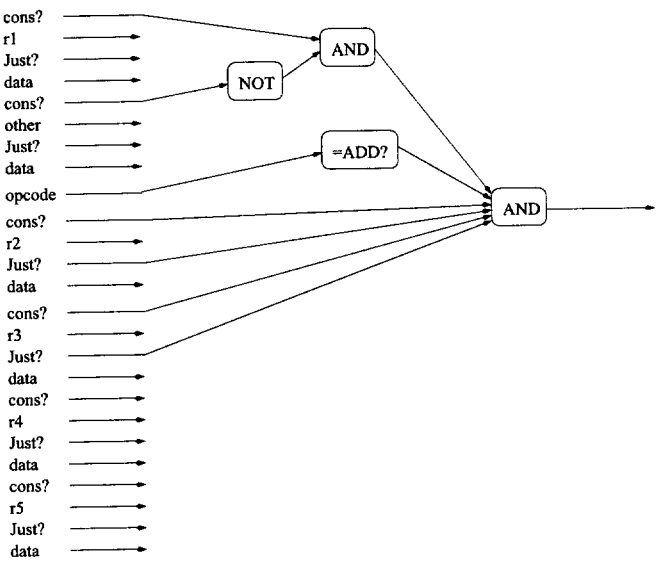
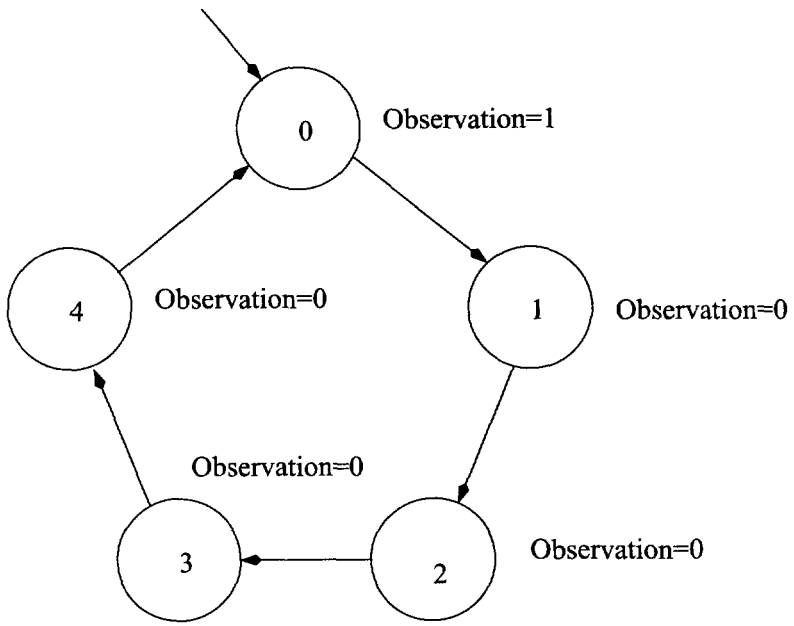


Figure 3.9: Schematic diagram for pattern matching circuit from `alu_stage`'s `ADD` case

The following example is a transition system that implements a modulo-5 counter, where the observation function returns a Boolean value that indicates when the system has been reset to 0:

$$\begin{aligned} \text{initial states} &: \{0\} \\ \text{next state relation} &: \{(x, y) \mid y = x + 1 \mod 5\} \\ \text{observation function} &: \{(0, 1)\} \cup \{(x, 0) \mid x \in \{1 \ldots 4\}\} \end{aligned}$$

Pictorially, this transition system can be drawn as the following graph:



In our programming language syntax we can define a type synonym, called `TS`, that uses the parameters `c`, `s`, `i`, and `o` to represent the set of transition systems with state-type `s`, input-type `i`, observation-type `o` and collection-type `c`:

```
TS c s i o = (c s, i -> s -> c s, s -> o)
```

The generality of the parameter `c` is not strictly necessary for this chapter, but is key in Chapters 5 and 6. We will typically restrict ourselves to finite transition systems with finitely non-deterministic transitions:

```
TS FSet s i o = (FSet s, i -> s -> FSet s, s -> o)
```

Notice that this ensures that the state of the transition systems remain finite.

Rather than explicitly naming the initial states and next-state relation within the transition system, we will often use the following projection functions on transition systems:

$$\begin{aligned} \texttt{initial (x,y,z)} &\triangleq \texttt{x} \\ \texttt{next (x,y,z)} &\triangleq \texttt{y} \\ \texttt{observe (x,y,z)} &\triangleq \texttt{z} \end{aligned}$$

We use the notation $a \xrightarrow[u]{i} a'$ as a shorthand for $a' \in (\llbracket \mathbf{next} \rrbracket\ u)\ i\ a$.

3.3 Specifying and modeling microarchitectures

In this section we demonstrate how ISA specifications and microarchitectural models are typically built. We construct an ISA specification and a microarchitectural model—both as transition systems. These transition systems will be used again in later chapters.

3.3.1 An example instruction-set architecture specification

For simplicity our specification does not support instruction or data memory. On each cycle the machine accepts an instruction as an input, reads the source register references, calculates the value of the destination operand and stores it into the register file. This transition system has type:

```
risc :: TS FSet Opcode (RF,Word) (Obs RF)
```

The type `RF` represents a register-file (which is essentially an environment or function from `Reg` to `Word`). We use the type `Reg` to represent register names, and `Word` to represent 32-bit words. If we expand the type `TS` out to:

```
( FSet (RF,Word)
, Opcode -> (RF,Word) -> FSet (RF,Word)
, (RF,Word) -> Obs RF
)
```

then we can see that `risc` is a triple, where the first element is a finite set of register-file and word pairs; the second element is a functional relation indexed by the elements of the

type `Opcode`, and the third element is a function from pairs of register-files and words to elements in the type `Obs RF`.

The observation-type `Obs`, defined in Figure 3.10, is used in `risc` in the following way:

- If the machine is stalled then the observation is `Nothing`. That is, because the transition system is stalled, it is in a state in which the register-file cannot be observed.
- If the machine is not stalled and also not flushed then the observation is `Just Nothing`. This is another situation where the transition system is in a state in which the register-file cannot be observed.
- If the machine is not stalled, and it is flushed, then the observation is `Just (Just rf)`, where `rf` is a function that represents the current state of the register-file.

We make use of the observation functions in Figure 3.10 to simplify the task of modeling with the `Obs` type. For example, we use `r_flushed` to construct the appropriate encoding with the type `Maybe` to model the case where the register-file is available for inspection.

The definition of `risc` is in Figure 3.11. The observation function (lines 14 through 16 of Figure 3.11) defines the significance of the integers in the initial states. If the integer is 1, the machine is flushed and the register-file is observable. If the integer is 2, the machine is not flushed, but also not stalled. If the integer is 3, the machine is stalled.

Note that `risc` is not pipelined itself. It does, however, implement an interface that allows for it to be used in environments built for pipelining: at any time the system can be stalled, flushed, or not stalled and not flushed. An implementation of this specification is, therefore, free to choose when the contents of the register-file are made available.

The next-state relation (lines 6 through 12 of Figure 3.11) is specified with transactions. On line 11, the function `make_trans` (Figure 3.3) is used to create a new transaction from the incoming instruction. The function `read_stage` takes the new transaction and fills in its source register operands. Then, `alu_stage` uses the source register values to calculate the value(s) of the destination operand(s). Finally, `wb_stage` updates the register file with the bindings from the destination operand(s).

```
type Obs e = Maybe (Maybe e)

r_stalled =  Nothing
r_not_flushed  = Just Nothing
r_flushed e = Just (Just e)

stall_obs :: Obs a  -> Bool
stall_obs o = case o of
                Nothing -> True
                otherwise -> False

flushed_obs :: Obs a  -> Bool
flushed_obs o = case o of
                  (Just (Just x)) -> True
                  otherwise -> False

stalling :: TS m i s (Obs  e) -> s -> Bool
stalling m s =  stall_obs (observe m s)

flushed :: TS m i s (Obs  e) -> s -> Bool
flushed m s = flushed_obs (observe m s)

view :: TS m i s (Obs  e) -> s -> e
view m s = case observe m s of
             (Just (Just x)) -> x
```

Figure 3.10: The type Obs, used to represent observations

```
0  risc :: TS FSet Opcode (RF, Int) (Obs  RF)
1  risc = (int, nxt, ob )
2     where
3
4     int = unit (initial_rf,1)
5
6     nxt i (rf,3) = either rf
7     nxt i (rf, n)
8            = let (rf', wbi) =  wb_stage wb rf
9                   wb = alu_stage rd
10                  rd = read_stage rf nw
11                  nw = make_trans i
12             in either rf'
13
14    ob (x, 1) = r_flushed x
15    ob (x, 2) = r_not_flushed
16    ob (x, 3) = r_stalled
17
18    either rf = do { x <- [1,2,3]; unit (rf,x) }
```

Figure 3.11: `risc`, a RISC transition system

3.3.2 An example pipelined RISC implementation

In the literature on formal verification, microarchitectural models are typically represented in the same manner as their specifications—albeit with the microarchitectural details filled in. In this section we define a microarchitectural-level implementation of `risc` that includes pipeline registers, and a next-state relation that defines how instructions flow through the registers, with possible stalling and bypassing.

Figure 3.12 provides the code for this three stage pipeline. In this transition system, the state of the pipeline is represented with a triple of transactions. The pipeline is flushed if each of the transactions is equal to `bubble_trans`, which is the transaction that represents the absence of action:

```
bubble_trans=([], BUBBLE , [])
```

The machine has only one initial state (defined on line 5): the initial register-file used in the definition of `risc`, paired with a flushed vector of transactions. The observation function (lines 14 through 16) determines whether or not the machine is flushed based on the vector of transactions.

Lines 7 through 12 define the next state relation, which implements a classic pipeline, with a register read stage, an ALU stage, and a write-back stage. The definition of the next-state relation is similar to `risc`'s next-state relation. The difference is that the intermediate values are stored in pipeline registers with appropriate bypassing.

Lines 14 through 16 define the observation function, which displays the register-file only when the pipeline is flushed.

3.4 Correctness criteria

Although ISA specifications and microarchitectural models are developed in the same formalism, they are typically developed at different levels of abstraction. For example, `risc` provides details on which values should be produced after executing a sequence of instructions, but does not specify when the values should be ready. The implementation, in contrast, is much more specific as to when the values are visible. These two systems are apparently related in some way—but how?

```
0  pipe :: TS FSet Opcode (RF, (Trans, Trans, Trans)) (Obs  RF)
1  pipe = (int, nxt, ob )
2     where
3     flushed_pipe = (bubble_trans,bubble_trans,bubble_trans)
4
5     int = unit (initial_rf, flushed_pipe)
6
7     nxt i (rf, (nw, rd, wb))
8           = let (rf', wbi) =  wb_stage wb rf
9                 wb' = alu_stage (bypass wbi rd)
10                rd' = bypass wbi (read_stage rf nw)
11                nw' = make_trans i
12            in unit (rf', ( nw', rd', wb'))
13
14    ob (rf, pipe)
15          = if pipe == flushed_pipe then r_flushed rf
16            else r_not_flushed
```

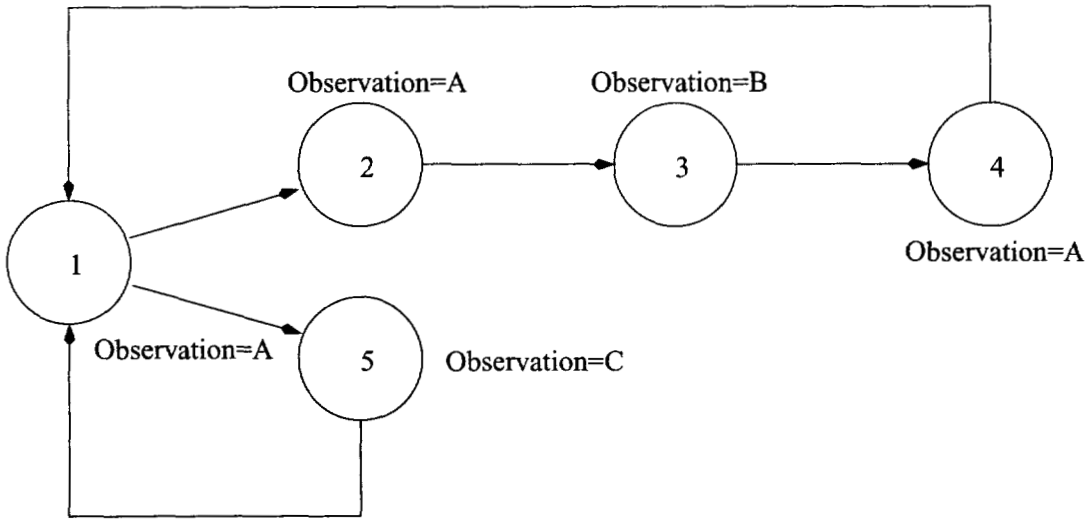
Figure 3.12: `pipe`, the RISC pipelined transition system

Typically, correctness is stated with a pre-order relationship such as *simulation* or *flush-point correctness*, which we define in the next section. Note that there are many variations of correctness criteria used in the literature that we will not review. Our purpose here is to discuss only those criteria necessary in the later chapters.

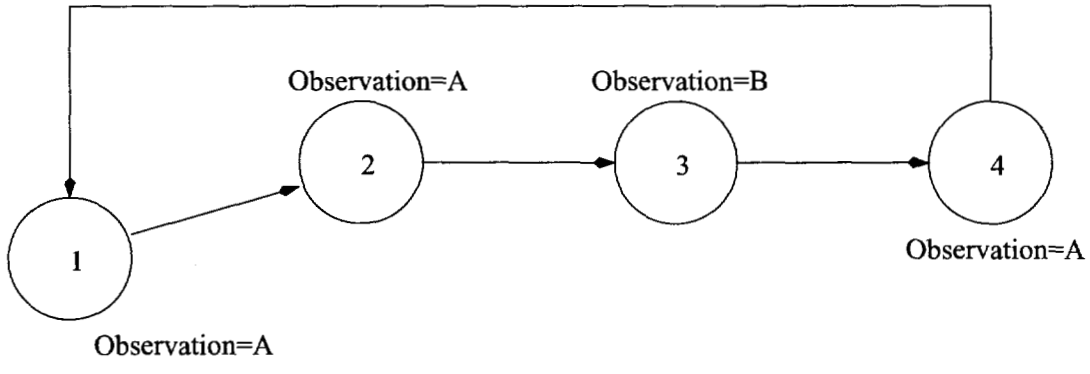
3.4.1 Simulation

A transition system u is said to *simulate* a specification v (notationally $(u, v) \in \mathsf{SIM}$) if there exists a relation R—called a simulation relation—that implies observational equivalence between sequences of u and v states. That is, R must hold between the initial states of the two systems and, for any step that the system u makes, v must have an analogous step that maintains R.

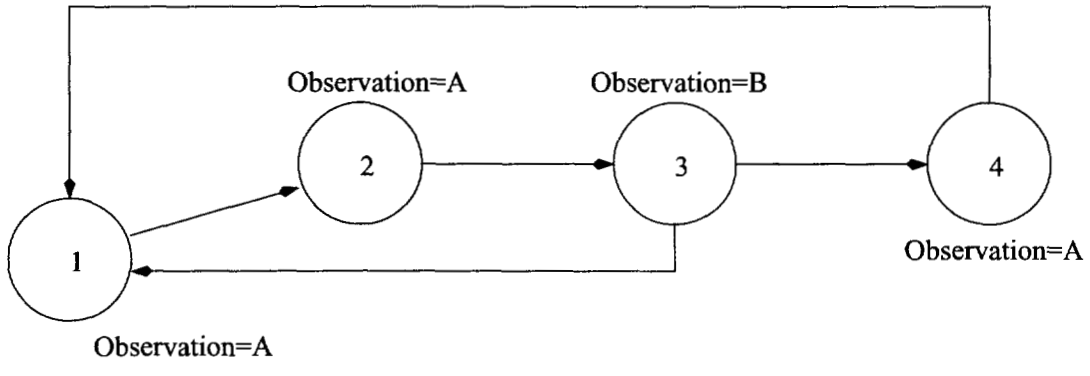
The intuition behind simulation is that the graph of executions in the implementation must be a subset of those of the specification. For example, imagine that the following graph represents the set of all possible executions in a specification:



Assume that the labels are from the observation function applied to the states. A system that simulates this specification must be a sub-graph of the above graph. For example, the graph below denotes a system which simulates the above system:



The next graph, on the other hand, is a system that does not simulate the specification:



Definition 3 (SIM) *Given* $(u :: \texttt{TS FSet}\ s\ i\ o)$ *and* $(v :: \texttt{TS FSet}\ s'\ i\ o)$, $(u, v) \in \mathsf{SIM}_R \triangleq$

SIM.A) $\forall a \in \texttt{initial}\ u.\ \exists b \in \texttt{initial}\ v.(a,b) \in R$

SIM.B) $\forall a, a', b, i.\ [(a,b) \in R \wedge a' \in \texttt{next}\ u\ i\ a] \Rightarrow [\exists b'.\ b' \in \texttt{next}\ v\ i\ b \wedge (a',b') \in R]$

SIM.C) $\forall a, b.\ (a,b) \in R \Rightarrow \texttt{observe}\ u\ a = \texttt{observe}\ v\ b$

In the example above, the transition system pictured in the first graph is simulated by the second graph, where

$$R = \{(n,n) \mid n \in \{1 \ldots 4\}\}$$

Note that, in this definition of SIM, we are abusing notation. To be more precise we should state condition SIM.C as:

$$\forall a, b.\ (a,b) \in R \Rightarrow [\![\texttt{observe}]\!]\ u\ a = [\![\texttt{observe}]\!]\ v\ b$$

It is again an abuse of notation to mix type constructors such as TS with mathematical variables such as i. A more precise use would be:

$$u :: [\![\texttt{TS FSet}]\!]\ s\ i\ o$$

We will use the less restrictive notation when there is no chance for ambiguity.

Often we use SIM without specifying a relation subscript. In that case we define

$$(u,v) \in \mathsf{SIM} \triangleq \exists R.\ (u,v) \in \mathsf{SIM}_R$$

We will sometimes refer to just the inductive clause of SIM: SIM.B

A common twist on simulation is the notion of *bisimulation.* If two systems are bisimular, then their graphs are isomorphic.

Definition 4 (BISIM)

$$(u,v) \in \mathsf{BISIM}_R \triangleq (u,v) \in \mathsf{SIM}_R \wedge (v,u) \in \mathsf{SIM}_{R^{-1}}$$

We can expand the definition of BISIM and define it directly as:

BISIM.A)

$$\forall a \in \texttt{initial}\ u.\ \exists b \in \texttt{initial}\ v.(a,b) \in R$$

and

$$\forall b \in \texttt{initial}\ v.\ \exists a \in \texttt{initial}\ u.(a,b) \in R$$

BISIM.B)

$$\forall a, a', b, i.\ [(a, b) \in R\ \wedge\ a' \in \texttt{next}\ u\ i\ a] \Rightarrow [\exists b'.\ b' \in \texttt{next}\ v\ i\ b\ \wedge\ (a', b') \in R]$$

and

$$\forall a, b, b', i.\ [(a, b) \in R\ \wedge\ b' \in \texttt{next}\ v\ i\ b] \Rightarrow [\exists a'.\ a' \in \texttt{next}\ u\ i\ a \wedge\ (a', b') \in R]$$

BISIM.C) $\forall a, b.\ (a, b) \in R \Rightarrow \texttt{observe}\ u\ a = \texttt{observe}\ v\ b$

In the microarchitecture verification literature, it is common to find simulation relations that are, in fact, functions. These functions are known by various names, including: abstractions, mappings, simulation mappings, abstraction mappings, or refinement mappings. We will refer to simulation with a mapping as MAP:

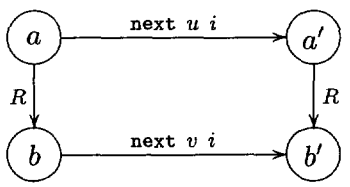
Definition 5 (MAP) *Given* $(u :: \texttt{TS FSet}\ s\ i\ o)$ *and* $(v :: \texttt{TS FSet}\ s'\ i\ o)$, $(u, v) \in \mathsf{MAP}_R \triangleq$

MAP.A) $\forall a \in \texttt{initial}\ u.\ \exists b \in \texttt{initial}\ v.\ R(a) = b$

MAP.B) $\forall a, a', b, i.\ [R(a) = b \wedge a' \in \texttt{next}\ u\ i\ a] \Rightarrow [\exists b'.\ b' \in \texttt{next}\ v\ i\ b \wedge R(a') = b']$

MAP.C) $\forall a, b.\ R(a) = b \Rightarrow \texttt{observe}\ u\ a = \texttt{observe}\ v\ b$

MAP.B is often presented pictorially as a commuting diagram:



3.4.2 Flush-point correctness

A difficulty with sophisticated microarchitectures is that it is rare for them to actually simulate their specifications. For instance, an out-of-order microprocessor will often execute a program in fewer cycles than the specification—meaning that the execution graph of the implementation cannot be a sub-graph of its specification.

A more flexible criterion, which we call flush-point correctness, has been independently proposed several times in the literature to address this weakness. It is general enough to allow machines which should be considered correct to be related.

Before we define flush-point correctness, we first must introduce several concepts used in its definition: the `Bubble` type-class, Γ, and the notion of a flush-point trace.

The Bubble type-class

We use the type-class `Bubble` to represent the existence of a machine instruction that takes no arguments:

```
class Bubble i where
    bubble :: i
```

The type `Opcode` is an instance of `Bubble`:

```
instance Bubble Opcode where
  bubble = BUBBLE
```

That is, when we use the value `bubble` in the context of the type `Opcode`, we are referring to the value `BUBBLE`. We define this type-class to facilitate the restriction of flush-point correctness to only those transition systems with a built in notion of a bubble instruction.

Removing bubbles with Γ

We use the notation (Γ *is*) to represent the set of finite instruction sequences that are equivalent to *is* when bubbles are not considered. That is, let $\flat$ remove all finite subsequences of bubbles from an instruction stream:

$$\Gamma\ is \triangleq \{is' | \flat\ is' = \flat\ is\}$$

We have borrowed this notation from Abadi and Lamport [4].

Flush-point traces

The final concept that we need is the notion of a flush-point trace, which is a finite sequence of states from a flushed state to the next flushed state. We will assume the existence of a predicate called flushed that indicates when the observation of the transition system is available. We define next_fp to represent the set of states that are at the end of the

flush-point traces from a given state a:

$$\begin{aligned}
\mathsf{next_fp}\ u\ i\ a\ \triangleq \{b \mid\ & \exists i' \in \Gamma\ i.\ \exists n.\exists s.\exists q \\
& |s| = n\ \wedge s_1 = a\ \wedge s_n = b \\
& \wedge \forall 0 \leq k < n - q.\ \mathtt{stalled}(s_k) \Leftrightarrow i'_k = \mathtt{bubble} \\
& \wedge \forall n - q \leq k < n.\ i'_k = \mathtt{bubble} \\
& \wedge\ \forall 0 \leq k < n.\ s_{k+1} \in [\![\mathtt{next}]\!]\ u\ i'_k\ s_k \\
& \wedge\ \mathsf{flushed}\ u\ a \\
& \wedge\ \mathsf{flushed}\ u\ s_n \\
& \wedge\ \forall 0 < k < n.\ \neg(\mathsf{flushed}\ u\ s_k) \\
\}&
\end{aligned}$$

Definition 6 *Given* $u :: \mathtt{TS\ FSet}\ s\ i\ (\mathtt{Obs}\ o)$, $v :: \mathtt{TS\ FSet}\ s'\ i\ (\mathtt{Obs}\ o)$, *Let*

$$(a, b) \in R^F \triangleq \mathsf{flushed}\ u\ a \wedge \mathsf{flushed}\ v\ b \wedge (a, b) \in R$$

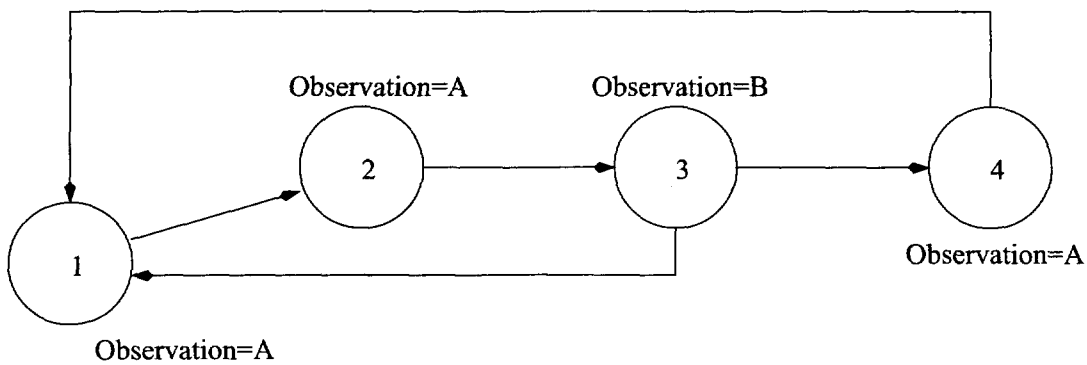
$(u, v) \in \mathsf{FP}_R \triangleq$

FP.A) $\forall a \in \mathtt{initial}\ u.\ \exists b \in \mathtt{initial}\ v.(a, b) \in R^F$

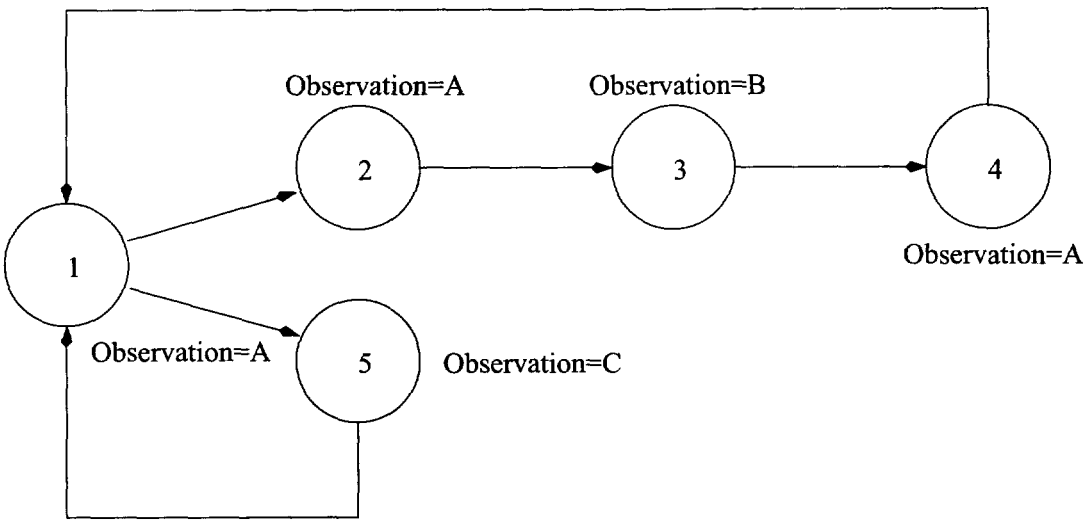
FP.B) $\forall a, a', b, is.\ (a, b) \in R^F \wedge a' \in \mathsf{next\_fp}\ u\ is\ a \Rightarrow \exists b'.\ b' \in \mathsf{next\_fp}\ v\ is\ b \wedge (a', b') \in R^F$

FP.C) $\forall a, b.\ (a, b) \in R^F \Rightarrow \mathtt{observe}\ u\ a = \mathtt{observe}\ v\ b$

The intuition behind flush-point correctness is similar to simulation, with the modification that we only compare states with R when they are flushed. When proving FP, we are essentially proving that there always exists a flush-point trace in the specification such that R holds at the points when the machines are in flushed states. Recall the example implementation from above:



And recall the specification that it does not simulate:



This implementation is correct with respect to FP if we let $R = \{(1,1),(3,3)\}$ and assume that only states 1 and 3 are flushed.

3.5 Verification methods

In this chapter we have seen how ISA specifications and microarchitectural models are represented. In addition, we have defined correctness criteria that relate specifications and models. But what are the common techniques used to prove these relationships? In this section we describe some of the most popular techniques used, including: *intermediate models*, *criteria strengthening*, *uninterpreted functions*, and Burch & Dill's *flush-based simulation mapping*. Later, in Chapter 6, we will see how these ideas are applicable to the proof of our WA microarchitectural model's correctness.

3.5.1 Intermediate models

A microarchitectural design that is being verified is usually very different from its ISA specification. It is often difficult to relate the components of the specification to the finely tuned reservation stations, arrays of execution units, branch predictors and content-addressable memories found in a sophisticated microarchitecture. Intermediate models are sometimes used in the literature to bridge this gap between microarchitectural models

and ISA specifications. These intermediate models are typically built using one of three concepts: *history variables*, *abstraction*, or *prophecy variables*.

History variables

Microarchitectural models typically only have registers that contribute to performance. However, when proving a relation between a microarchitectural model and its specification, it can be helpful for the microarchitectural model to carry around more state than is necessary. An intermediate model sometimes can be constructed with extra variables used to carry around history which clearly bisimulates the original microarchitectural model.

Abstraction

Intermediate models are also commonly used to abstract away the complexity of microarchitectural models. That is, an abstract model can be constructed from the microarchitecture with a simpler but less deterministic next-state relation. The original next-state relation can sometimes be proved to be correct with respect to the abstract one. When using this strategy, one must also show that the intermediate abstraction implements the specification.

Proposition 1

MAP, SIM, FP, MAP.B, SIM.B, and FP.B are pre-orders.

The preorder property is important, because it allows us to use this technique of introducing intermediate models. For example, imagine trying to prove $(A, B) \in \mathsf{SIM}$. We might build a new system B', which is based on B such that we can prove $(A, B') \in \mathsf{SIM}$. If we can prove $(B, B') \in \mathsf{SIM}$, then by transitivity of SIM we know $(A, B) \in \mathsf{SIM}$.

Prophecy variables

Some microarchitectural models perform computations faster, or make non-deterministic choices at times that are different than their specifications—making it impossible to prove SIM. In some cases an intermediate model can be built from a microarchitectural model that calculates the same answer as the original microarchitecture, but displays it at a

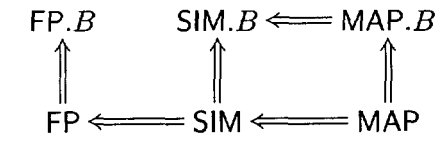


Figure 3.13: Connections between the correctness criteria

time which is more convenient when proving simulation. One way to address this problem is to use prophecy variables [4], which can slow down a microarchitectural model that retires too many instructions per transition or which can be made to predict the future. Let u be a microarchitectural model and v be an architectural specification. To show that $(u, v) \in \mathsf{FP}$, one might construct a slower model u' using a prophecy variable and show that $(u, u') \in \mathsf{FP}$. Then, after showing that $(u', v) \in \mathsf{SIM}$, by the transitivity of FP and $\mathsf{SIM} \subset \mathsf{FP}$ (when the domain of systems is restricted to initially flushed systems), $(u, v) \in \mathsf{FP}$.

Definition 7 *A system u is* initially flushed *if* $\forall s \in \texttt{initial}\ u.\ \mathsf{flushed}\ u\ s$

Unless stated otherwise, we will assume that all transition systems are initially flushed.

3.5.2 Criteria strengthening

Another proof strategy that is often used in unison with intermediate variables is the strengthening of the criteria. For example, we might prove SIM in order to prove FP. Figure 3.13 is a picture demonstrating the connections between the criteria mentioned in this chapter. For example, $\mathsf{SIM}.B \Leftarrow \mathsf{MAP}.B$ states that if we have proved $\mathsf{MAP}.B$ then we have also proved $\mathsf{SIM}.B$. Unfortunately, because $\mathsf{SIM}.B \not\subseteq \mathsf{FP}.B$, there is an arrow missing. In this figure we assume that domain is restricted to initially flushed systems. To see why $\mathsf{SIM}.B \not\subseteq \mathsf{FP}.B$, consider the following example:

System u System v

In this example, we have two transition systems—and each transition system has one state: a and b. The arrows indicate that the systems can transition to the same state. Assume that flushed u a, $\neg$flushed v b, and $R \triangleq \{(a, b)\}$ meets the criterion of SIM.B. There is a flush-point trace in u:

$$a \xrightarrow[u]{} a \xrightarrow[u]{} a \xrightarrow[u]{}$$

But there does not exist a flushed state in v. Therefore, there cannot be a flush-point trace in v.

Proposition 2 *If* $(u, v) \in$ SIM *and* u *is initially flushed then* $(u, v) \in$ FP

Proof. We know by SIM that there exists a simulation relation R.

A) Because u is initially flushed and SIM.A

B) Assume that a_0 and a_n are flushed states of u and that

$$a_0 \xrightarrow[u]{i_0} a_1 \xrightarrow[u]{i_1} \cdots \xrightarrow[u]{i_{n-1}} a_n \xrightarrow[u]{i_n}$$

is a flush-point trace. We know that $i_0 \ldots i_{n-1}$ meets the constraints required by next_fp. By SIM.B we know that there exists a sequence:

$$b_0 \xrightarrow[v]{i_0} b_1 \xrightarrow[v]{i_1} \cdots \xrightarrow[v]{i_{n-1}} b_n \xrightarrow[v]{i_n}$$

such that $(a_k, b_k) \in R$ for all $0 \leq k \leq n$. By SIM.C we know also that both b_0 and b_n are flushed.

C) By R and SIM.C.

□

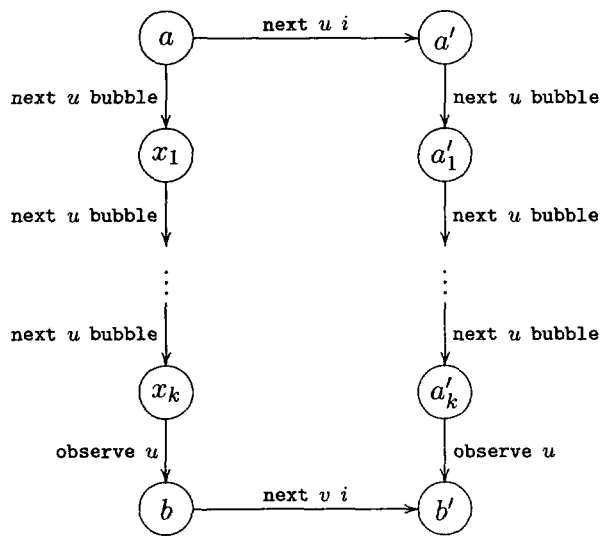
3.5.3 Uninterpreted functions

One of the most pervasive techniques in microarchitecture verification is the use of uninterpreted functions [12, 17, 43, 51, 63]. Suppose that both `next` u and `next` v are defined in terms of a function. When proving a relationship between expressions containing `next` u and `next` v, the interpretation of the function is often inconsequential. That is, it is commonly sufficient to know that applications of the function to equal arguments delivers equal results.

For example, Burch & Dill [17] defined both their architectural specification and pipelined model to access state using the functions `read` and `write`. When proving the correctness of a simulation mapping, they were able to do so without interpreting the meaning of `read` and `write`.

3.5.4 Calculating simulation mappings with flushing

Burch & Dill observed that a simulation mapping can be automatically calculated for pipelined microarchitectures that support the notion of bubble [17]. Assume that u :: `TS FSet s i o` is a microarchitectural model and v :: `TS FSet o i o` is an architectural specification. Assume that i is an instance of `Bubble`. If, for all states, there exists a k such that u is flushed after k applications of `next` u `bubble`, and the following diagram commutes for all i, then $(u, v) \in \mathsf{MAP}.B$:



In this case, the simulation mapping is $\texttt{observe}\ u \circ (\texttt{next}\ u\ \texttt{bubble})^k$.

3.6 Historical Context

To provide more context to the material in this chapter, we close with a history of the key technical developments in microarchitectural verification. Note that this historical summary only covers the literature up until 2002[1]. We survey the literature on the formal verification of out-of-order microarchitectural designs, and point out how the material developed in this chapter has been applied in practice. This section is based on a paper by Aagaard, Cook, Day and Jones [1, 3].

3.6.1 Burch: Verifying Superscalar Microprocessors

In some of the first work on processor verification that went beyond pipelining, Burch [16] proved that MAP.B holds between a two-instruction wide superscalar microarchitectural model and an instruction-set architecture. For this model, proving the commuting diagram with Burch & Dill's simulation mapping was not feasible using algorithmic methods. To address this, the paper presented four commuting diagrams that together imply MAP.B. The proof obligations in Burch's decomposition were significantly simpler and were feasibly solved with an automated decision procedure.

3.6.2 Damm, Pnueli, Arons: Verification by Refinement

In the first work to address the verification of out-of-order microarchitectural designs, Damm & Pnueli [20] proved that SIM holds between an out-of-order model and an ISA specification. The implementation and ISA were extraordinarily simple: neither handled branch or jump instructions, and both required finite instruction sequences. In addition, the implementation did not allow multiple instruction issue or retirement and the pipeline was abstracted away. Nevertheless, the proof was a significant accomplishment. In the proof, an abstract intermediate model was constructed that computes all out-of-order, data-flow executions. Uninterpreted functions were used to abstract the data paths. The

[1]When this dissertation research was done

intermediate model contained an arbitrary number of functional units. In later work, Arons & Pnueli [8, 9] used an intermediate model which used a prophecy variable for each instruction. The intermediate model and the specification were synchronized at instruction dispatch by comparing the instruction result in the specification with the prophecy variable in the intermediate model. Other proof steps established that the result written into the implementation's register file was the same as the prophecy variable.

Arons & Pnueli later extended their approach to an out-of-order model with a reorder buffer [7]. The systems were synchronized at instruction retirement. To facilitate this, the specification executed instructions only when they were retired.

3.6.3 Sawada & Hunt: Micro-Architecture Execution Trace Table

Sawada & Hunt [55] proved that FP.C held between an out-of-order model and specification ISA. Their implementation issued a single instruction per cycle. The out-of-order model had a number of execution units, including instruction and data memory, all of different latencies. The model did not use a reorder buffer. Correctness was proved by constructing an intermediate model that captured the history of all previously executed instructions.

In a later paper [56] the authors extended their approach to verify a model that handled precise exceptions, interrupts, and speculative branch execution with a reorder buffer. Hunt & Sawada again used the FP.B correctness criteria. Correctness was proved in the same manner as before. The intermediate model was adapted to carry relevant information about interrupts and exceptions.

3.6.4 Skakkebaek, Jones, & Dill: Incremental Flushing

Skakkebaek, Jones & Dill [60] proved FP.B between an ISA and a pipelined, out-of-order design. The implementation contained multiple execution units, and a reorder buffer. They introduced the notion of *incremental flushing*, the decomposition of the flushing-based abstraction into a collection of easier proof obligations. The intermediate model used in this proof placed the logic from multiple stages of the pipeline into a single stage that computed the result of each instruction when it entered the machine. The instruction

input was queued in the reorder buffer until retirement. The implementation and the intermediate model were proved to be in MAP.B. The intermediate model was proved to be in FP.B with the instruction-set architecture using incremental flushing.

In later work, Jones, Skakkebaek & Dill, [45, 46, 41] built a prophecy-variable based non-deterministic intermediate model in which the scheduling logic was abstracted. Incremental flushing was used to show that the machine executing at full capacity was in FP.B with an intermediate model that was restricted to execute only one instruction at a time. The intermediate model was then shown to be in the FP.B relation with the ISA.

3.6.5 Berezin, Biere, Clark & Zhu: Reference Files

Berezin, Biere, Clarke & Zhu [13] proved MAP.B between an out-of-order model and an ISA. An intermediate model was constructed with a *reference file*, a heap-like structure with sharing of common sub-expressions. MAP.B was proved between the intermediate model and the original implementation. MAP.B was proved between the intermediate model and the ISA, using Burch & Dill's simulation mapping. With a transitive argument, using the correctness hierarchy, the implementation was proved to be in MAP.B with the ISA.

3.6.6 McMillan: Compositional Model Checking

In contrast with the other work surveyed, McMillan [51] did not directly prove a criteria akin to SIM or FP. Instead he proved that an intermediate model with history variables satisfied several invariants [52]. However, we conjecture that McMillan's invariants imply SIM.B.

The proof was decomposed into three proof obligations. The first stated that the operand forwarding logic worked correctly; the second demonstrated that the instruction execution logic functioned correctly using uninterpreted functions. A third obligation extended the proof to a model with an arbitrary number of execution units.

3.6.7 Hosabettu, Gopalakrishnan, Srivas: Completion Functions

Hosabettu, Gopalakrishnan & Srivas [31, 34] verified the correctness of an out-of-order model. The important contribution of this paper was a powerful technique for decomposing proof obligations. This technique appears to be related to the programming languages notation of a *continuation*. The simulation mapping for a proof of MAP.C was computed by composing *completion functions* together, one for each instruction outstanding in the machine. A completion function represents the intended effect of an instruction on microprocessor state when it has completed. In subsequent papers, the completion functions approach was extended to an out-of-order machine with an unbounded reorder buffer [32] and a similar machine without a reorder buffer [33]. Both extensions used MAP.B and an intermediate model with auxiliary variables to establish FP.B.

In another paper [30] the authors extended their approach to microarchitectures with branch speculation and instruction exceptions. The notion of completion functions was again extended, this time to include the possibility that an instruction may be speculative, and thus never complete. FP.C was again established by proving MAP.B with an intermediate model.

3.6.8 Velev & Bryant: Exploiting Positive Equality

Velev & Bryant [64] extended Burch & Dill's approach to accommodate superscalar microarchitectures. They constructed an intermediate model with an efficient reduction of the original logic to propositional logic. This reduction was facilitated by exploiting *positive equality*, a technique that efficiently handles terms that appear in a positive polarity.

In a subsequent paper the authors extended their approach to microarchitectures with arbitrary-latency functional units and memories, branch prediction, and instruction-generated exception handling [65].

3.7 Conclusion

In this chapter we have developed a transition system formalism along with correctness criteria used to relate transition systems. We have discussed several common techniques

used to decompose and simplify microarchitectural correctness proofs. We have also surveyed the literature on microarchitectural verification—pointing out which criterion was proved and which structuring techniques were used during the proof.

As we have learned in this chapter, transition systems are the fundamental formalism used to describe both microarchitectural models and ISA specifications. The correctness criteria proved between models and specification are typically variants of simulation or flush-point correctness. Uninterpreted functions, intermediate models, Burch & Dill's simulation mapping and the correctness criteria hierarchy are some of the most pervasive techniques used to handle the complexity inherent in proving modern microprocessor microarchitectural models correct.

In the subsequent chapters we will use the material described here as a basis for further development. For example, we will discuss how to extend the modeling techniques from Section 3.2 and the correctness criteria from Section 3.4.

Chapter 4

Modeling with transformers

In this chapter we develop a method of modeling architectural extensions in a modular fashion. We use this method to build specifications of our extended instruction-sets, WA and OA, and simple microarchitectural models that implement them. The microarchitectural model of WA implements a pipeline that integrates predication. The OA implementation provides three pipelined execution clusters. We make use of these models later in the dissertation when we discuss how to formally prove models of this style correct.

As is standard practice, we specify WA and OA as transition systems. However, rather than monolithic transition systems, they are represented as the composition of functions. The specification of WA is defined as:

```
wa p = fnt p (prd risc)
```

The functions `fnt`, and `prd` take transition systems as arguments and return transition systems; each potentially with more features. These functions could specify an extension such as instruction caching, parallelism or predication. We call these functions *transition system transformers.*

The transition system `risc` is described in detail in Chapter 3 and is displayed in a visual representation in Figure 4.1. In the specification of WA, `risc` is extended to support predication by the transformer `prd`, as seen in Figure 4.3. The transformer `fnt` builds a transition system with a program memory loaded with the program `p`, as shown in Figure 4.2.

A WA microarchitectural model can also expressed as the composition of transformers:

```
wma p = fnt p prd_pipe
```

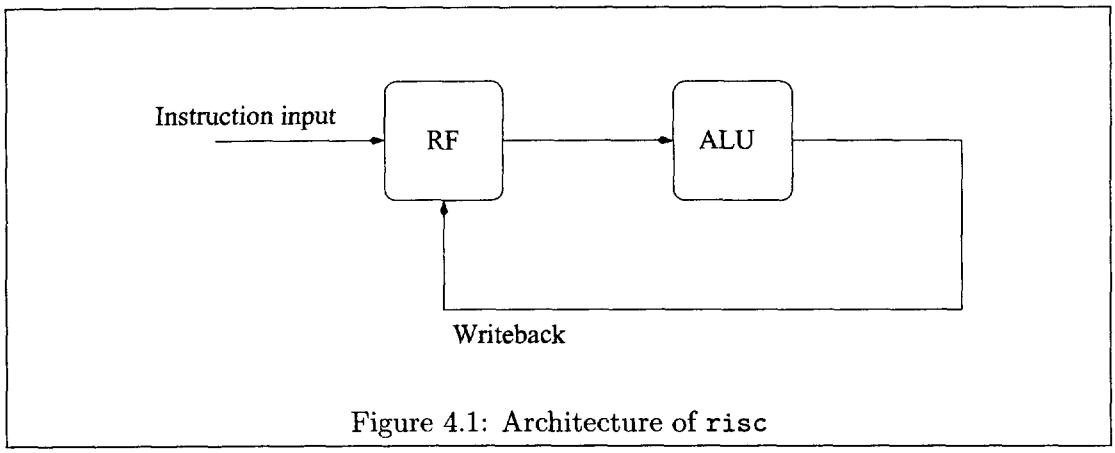


Figure 4.1: Architecture of `risc`

This model is, in principle, similar to what we might find in the Intel StrongARM microprocessor. It is a fast predicated pipeline, with an instruction fetch unit on the front end. Figure 4.5 displays the WA microarchitectural model. Figure 4.4 displays the WA architectural model.

In order to specify the OA we can extend WA with concurrency as follows:

```
oa p = fnt p (cnc 1 (prd risc))
```

The parallelism transformer `cnc` then adds a notion of threads, thread identifiers, and synchronization—see Figure 4.6. `cnc` can also be used in the construction of a microarchitectural model for OA:

```
oma p = fnt p (cnc 3 prd_pipe)
```

The transformers `fnt` and `cnc` are shared between the specification and implementation. However, in the implementation, `cnc` is applied to 3 instead of 1, which means that `cnc` constructs 3 concurrently executing transition systems. The transition system `prd_pipe` is a predicated pipeline with better stalling behavior than `prd pipe`. See Figure 4.8 for a visual representation of the microarchitectural system.

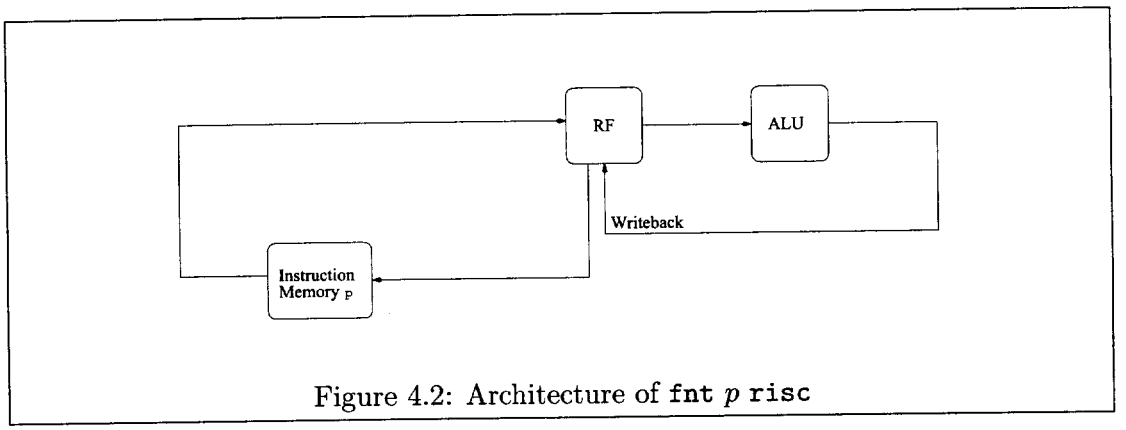


Figure 4.2: Architecture of `fnt` p `risc`

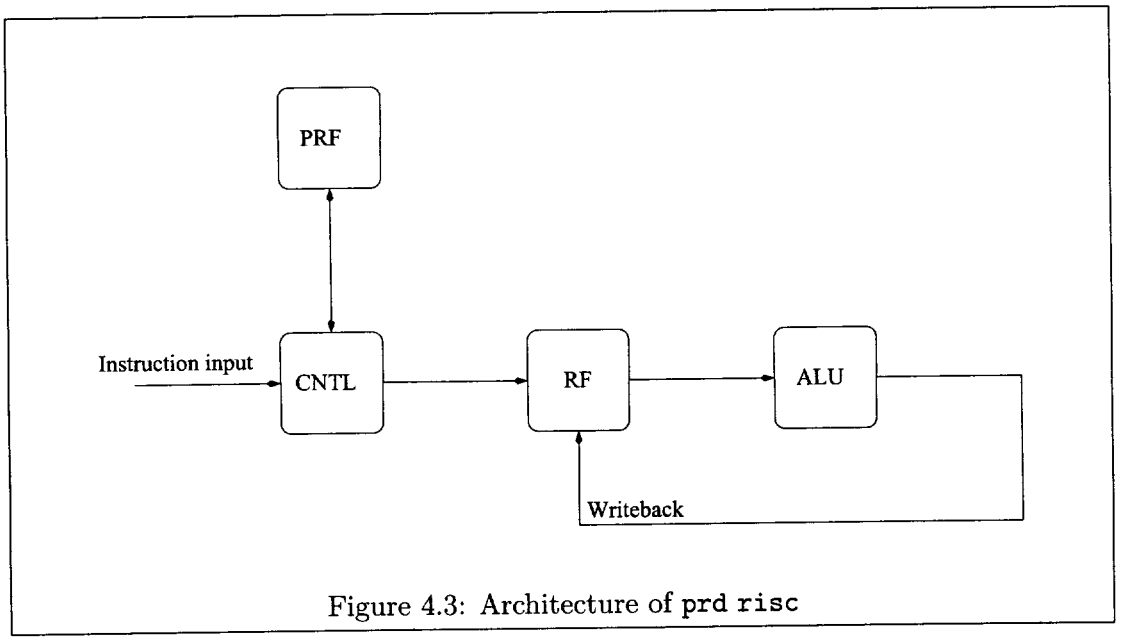


Figure 4.3: Architecture of `prd risc`

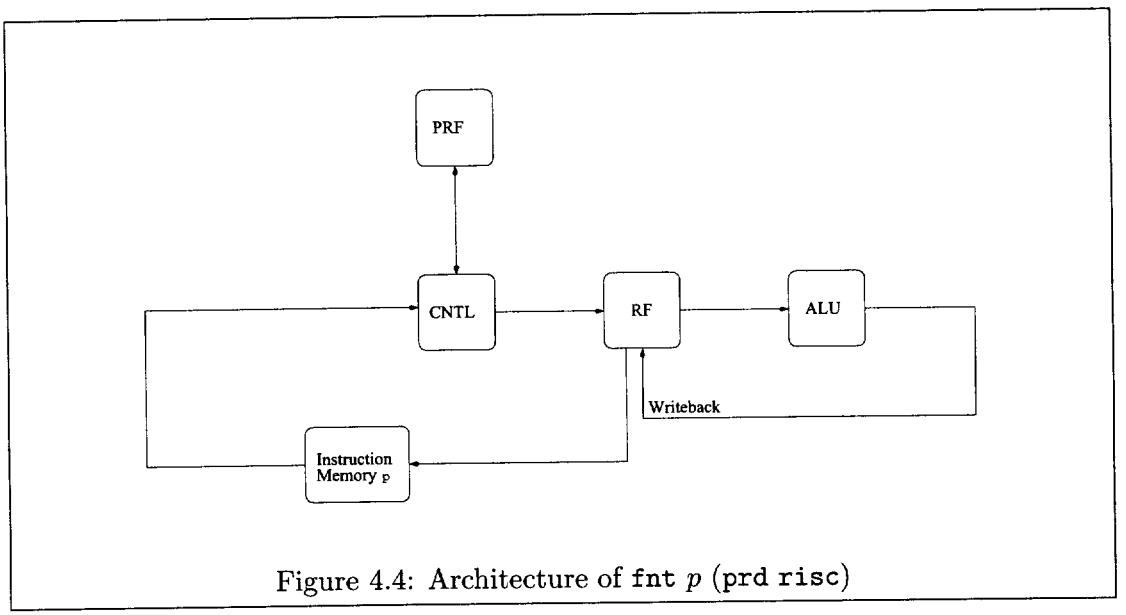


Figure 4.4: Architecture of `fnt` p (`prd risc`)

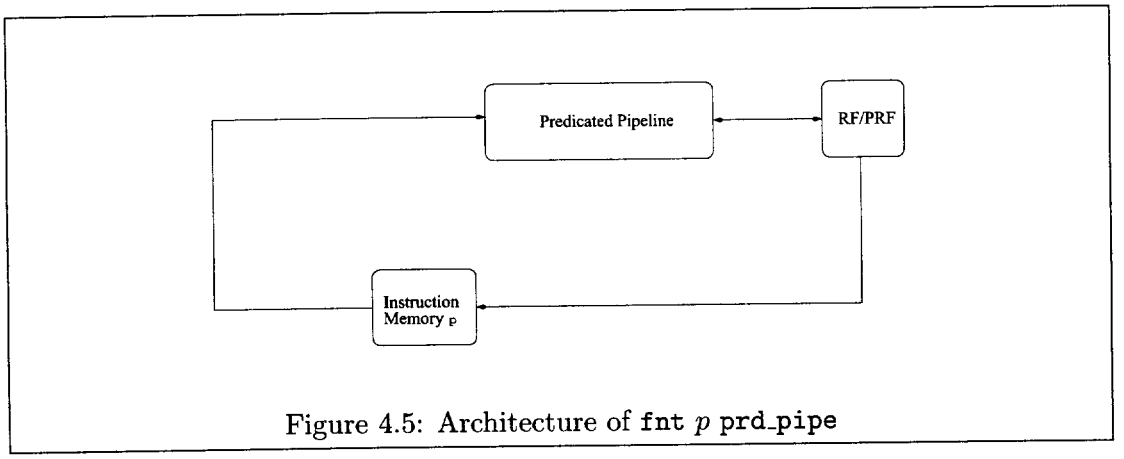


Figure 4.5: Architecture of `fnt` p `prd_pipe`

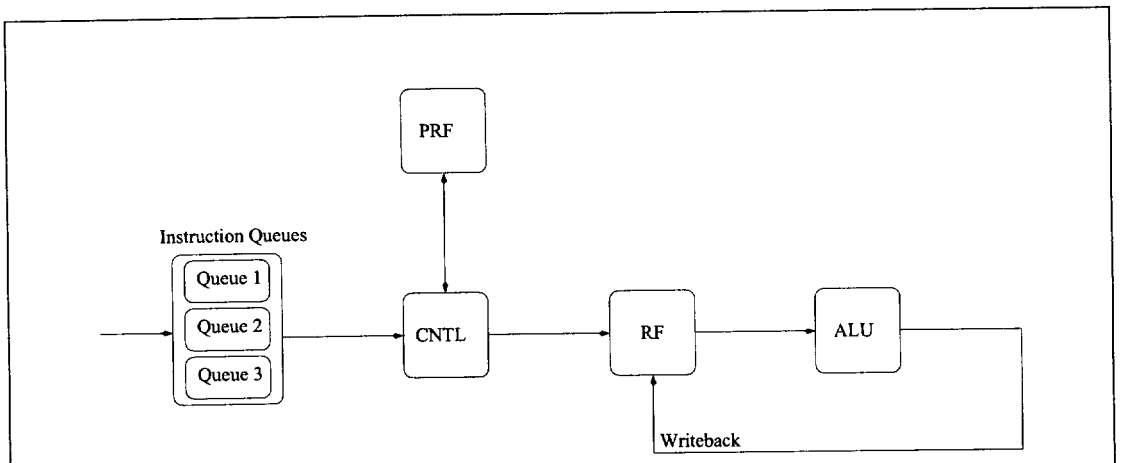


Figure 4.6: Architecture of `cnc 1 (prd risc)`

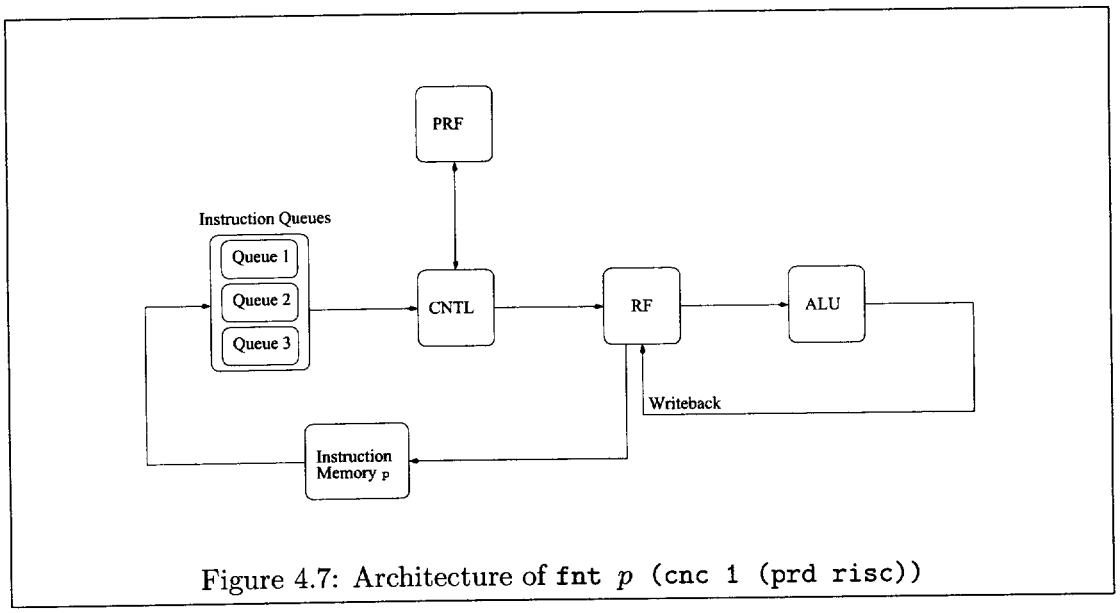


Figure 4.7: Architecture of `fnt` p `(cnc 1 (prd risc))`

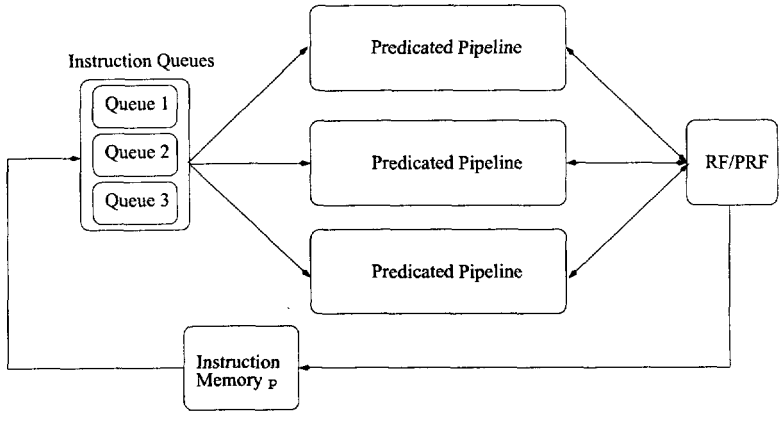


Figure 4.8: Architecture of `fnt` p `(cnc 3 prd_pipe)`

4.1 Predicating the architectural model

The transformer `prd` is designed to construct a predicated transition system. This transformer has type:

```
(Collect c, Bubble i, Eq r, Eq w, Bind i r w, Integral w) =>
    TS c i s (Obs (Env r w)) ->
    TS c (Prd_Instr i r r) (Prd_St s r (Prd_Instr i r r)) (Obs (Env r w))
```

In other words, the transformer takes a transition system whose only constraint is that it have observation type: `Obs (Env r w)`, meaning that the transition system provides a register-file as its observation type, but only when it is flushed and not stalled. It returns a transition system with the same observation type, but with richer types representing the inputs and states. The returned input-type is `Prd_Instr i r r`. `Prd_Instr` (Figure 4.9), when given an instruction type `i`, register-type `r` and predicate register type `r'`, returns a type that represents tagged expressions. These tagged expressions can be either register to predicate-register moves (`R2P`), predicate-register to register moves (`P2R`), an assignment to the predicate register file (`SET`), a predicated instruction (`IF`), or an un-predicated instruction (`GO`).

```
data Prd_Instr i r r' = R2P r' r' r | P2R r r' | SET r' Bool
                      | IF i r' | GO i

instance Bubble i => Bubble (Prd_Instr i r r') where
        bubble = GO bubble
```

Figure 4.9: Predicated instruction type and instances

The returned state-type is `(s,(Env r Bool,Maybe i))`. The type `Env r Bool` is used to represent a predicate register-file. Type type `Maybe i` is used to represent the instruction register. For clarity, we abbreviate this type with the synonym `Prd_St`:

```
type Prd_St s r i = (s,(Env r Bool,Maybe i))
```

The occurrence of the type-class `Bind` in the type of `prd` asserts that we need to provide a function named `bind` for `i`, `r`, and `w`:

```
class Bind i r w where
        bind :: r -> w -> i
```

That is, an expression such as `bind r10 20` should form an instruction that directs the transition system to bind the register `r10` to 20 in the register file. In the case of `Opcode`, `bind` is defined as `SET`.

The definition of the transformer `prd` is in Figure 4.10. The set of initial states of the system (defined on line 2) are essentially the initial states of the underlying system paired with a predicate register-file and an empty register. The observation function is defined in the equations on lines 4 and 5. On line 4, in the case that there is an instruction in the register, the observation is that the system is stalled. Line 5 states that for any given state, when the register is empty, the observation is simply the observation of the underlying state.

Lines 8 through 36 define `prd`'s next-state relation. For example, lines 8 through 12 define what the system does when there is an instruction (`i`) in the register. The next-state relation checks to see if the underlying system is flushed. If so, then the saved instruction

```
0  prd m = (int,nxt,ob)
1      where
2      int = do {x <- initial m; unit (x,(emptyEnv True,Nothing))}
3
4      ob (s,(e,Just _)) = r_stalled
5      ob (s,(e,Nothing)) = observe m s
6
7
8      nxt _ (s,(e,Just i)) = if flushed m s
9                                then nxt i (s,(e,Nothing))
10                               else do { s' <- next m bubble s
11                                       ; unit (s',(e,Just i))
12                                       }
13     nxt (P2R r r') (s,(e,Nothing))
14        = if (readEnv e r') then do { s' <- next m (bind r 1) s
15                                    ; unit (s',(e,Nothing))
16                                    }
17          else do { s' <- next m (bind r 0) s
18                  ; unit (s',(e,Nothing))
19                  }
20     nxt (SET r' b) (s,(e,Nothing))
21                           = do { s' <- next m bubble s
22                                ; unit (s',(updateEnv e (r',b),Nothing))
23                                }
24     nxt (IF i r') (s,(e,Nothing)) = if readEnv e r'
25                 then do { s' <- next m i s; unit (s',(e,Nothing))}
26                 else do { s' <- next m bubble s; unit (s',(e,Nothing))}
27     nxt (GO i) (s,(e,Nothing)) =
28          do { s' <- next m i s; unit (s',(e,Nothing))}
29     nxt (R2P r1 r2 r') (s,(e,Nothing))
30       = if flushed m s then
31              let e1 = updateEnv e (r1,readEnv (view m s) r' /= 0)
32                  e2 = updateEnv e1 (r2,readEnv (view m s) r' == 0)
33              in unit (s,(e2,Nothing))
34         else do { s <- next m bubble s
35                 ; unit (s,(e,Just (R2P r1 r2 r')))
36                 }
```

Figure 4.10: The predication transformer **prd**

is issued. If not, then a bubble is placed into the underlying system and the instruction is kept in the register.

In the case that there is no instruction in the register, the next-state relation performs the appropriate action for each instruction type. If the incoming instruction is an `IF` (line 24), and the value of the predicate in the predicate register-file is true, then the instruction is passed on to the underlying system. In the case that the predicate is untrue, then a `bubble` is placed into the underlying system in the instruction's place.

4.2 Predicating the microarchitectural pipeline

To build a predicated pipeline we could simply apply `prd` to the transition system `pipe` from Chapter 3. Unfortunately, because `prd` places the predication check at the front of the pipeline, it must flush the pipeline each time it encounters an `R2P` instruction. For example, consider executing the following code fragments in `prd pipe`:

```
[ GO (ADD r1 r2 r3)
, GO (ADD r4 r5 r6)
, GO (ADD r7 r8 r9)
, R2P pr1 pr2 r7
]
```

In this case, `prd` would issue the first three add instructions and then flush the pipeline before issuing the `R2P` instruction. That is: the predication transformer would cause the machine to stall and drain the add instructions before copying the value of `r7` into the predicate register-file. The result of this would be a three cycle penalty for each `R2P` instruction. For this reason, we will not use it in the definition of the microarchitecture. It will, however, turn out to be useful later when decomposing the proof of the implementation's correctness.

To build a faster predicated pipeline, the aspects of `prd` must be integrated into the system such that instructions are allowed to flow through the pipeline before the predication check occurs. Each instruction's predicate can then be checked just before writeback.

The integrated predication pipeline, called `prd_pipe` (Figure 4.11), implements this algorithm. It is constructed using transaction combinators which are a hybrid of the

```
0  prd_pipe :: TS FSet PInstr
1              ((RF,(Prd_Trans, Prd_Trans, Prd_Trans)),(PRF,()))
2              (Obs  RF)
3  prd_pipe = (int, nxt, ob )
4     where
5     flushed_pipe = (prd_bubble_trans,prd_bubble_trans,prd_bubble_trans)
6
7     int = unit ((initial_rf, flushed_pipe), (initial_prf, ()))
8
9     nxt i ((nrf, (nw, rd, wb)), (prf, other))
10           = let rf= (nrf, prf)
11                 (rf', wbi) =  prd_wb_stage wb rf
12                 wb' = prd_alu_stage (prd_bypass wbi rd)
13                 rd' = prd_bypass wbi (prd_read_stage rf nw)
14                 nw' = prd_make_trans i
15             in  unit ((fst rf', ( nw', rd', wb')), (snd rf', other))
16
17    ob ((rf, pipe), (prf, other))
18           = if pipe == flushed_pipe then r_flushed rf
19             else r_not_flushed
```

Figure 4.11: prd_pipe: a higher-performance predicated RISC pipeline

transaction combinators from Chapter 3. The transition system is defined to manipulate pipeline stages of type `Prd_Trans` (Figure 4.12) which is like `Trans`, except that predicate registers are possible source and destination operands. One can think of an element of `Prd_Trans` as two transactions, with a shared instruction. This is the essence of how the combinators treat `Prd_Trans` instructions. For example, the function `prd_bypass` defined in (Figure 4.13) bypasses the predicate references and the standard references independently. The writeback function, `prd_wb_stage`, (Figure 4.14) checks the source predicate and commits the instruction only if the predicate is true.

Line 5 of Figure 4.11 defines how an empty stage is represented internally within `prd_pipe`. Line 7 declares that there is a single initial state, using the variables `initial_rf` and `initial_prf` as representatives of initial register-file states. The next state relation is defined in a fashion similar to the definition of the `pipe`, except that the predication-based transaction functions are used.

```
type Prd_Trans = ( [(Reg,Maybe Int)] , [(PReg,Maybe Bool)]
                 , Prd_Instr Opcode Reg PReg
                 , [(Maybe Reg,Maybe Int)] , [(Maybe PReg,Maybe Bool)]
                 )
```

Figure 4.12: Prd_Trans: the predication transaction type

```
prd_bypass :: Prd_Trans -> Prd_Trans -> Prd_Trans
prd_bypass (dsts1,dsts2,i,srcs1,srcs2) (dsts1',dsts2',i',srcs1',srcs2')
       = (dsts1',dsts2',i',srcs1'',srcs2'')
     where (_,_,srcs1'') = bypass (dsts1,i,srcs1) (dsts1',i',srcs1')
           (_,_,srcs2'') = bypass (dsts2,i,srcs2) (dsts2',i',srcs2')
```

Figure 4.13: pred_bypass: the predication bypass combinator

```
prd_wb_stage :: Prd_Trans -> MEnv -> (MEnv, Prd_Trans)
prd_wb_stage (a,b,IF c d,e,f) env
  = let t = (a,b,IF c d,e,f)
    in if pred_true t then (do_wb  t env, t)
       else (env, prd_bubble_trans)
prd_wb_stage  t env =  (do_wb  t env, t)

do_wb :: Prd_Trans -> MEnv -> MEnv
do_wb  (dsts1,dsts2,i,srcs1,srcs2) (rf,prf) = (rf',prf')
   where rf' = foldl writeback rf dsts1
         prf' = foldl writeback prf dsts2
         writeback rf (r,Just x) = updateEnv rf (r,x)

wb_stage :: Trans -> RF -> (RF,Trans)
wb_stage  (dsts1,i,srcs1) rf = (rf',(dsts1,i,srcs1))
   where rf' = foldl writeback rf dsts1
         writeback rf (r,Just x) = updateEnv rf (r,x)
```

Figure 4.14: wb_stage: the predication writeback function

```
emptyRn :: Int -> Region i
isEmptyRn :: Region i -> Bool
popRn :: Int -> Region i -> (Maybe i,Region i)
```

Figure 4.15: Interface to the `Region` type

4.3 Adding concurrent execution

The concurrency transformer `cnc` is used both in the architectural specification and microarchitectural model. The idea behind this transformer is that it takes a transition system and makes a number of concurrently executing copies of it. These copies have some shared state.

The type of the concurrency transformer is:

```
Bubble i => Int -> TS FSet i ((v,o),s) (Obs  e) ->
            TS FSet (Region i) ((v,s),[o],Region i) (Obs  e)
```

The first parameter is used to determine the number of concurrently executing transition systems. The second parameter is a non-deterministic transition system. The transformer returns a new nondeterministic transition system with richer input- and-state types. The input-type of the returned transformer is regions of inputs.

The state space of the new system is a triple `((v,s),[o],Region i)` :

- The first element is a pair from the underlying system's state space. This is the shared state.
- The second element is a list of parts from the underlying system's state space—one for each concurrently executing transition system.
- The third element represents the state of the current region in execution.

The `Region` data-type (Figure 4.15) is used to represent basic blocks, as described in Chapter 2. Essentially, a `Region` represents a sequence of queues. Figure 4.15 contains the types and names of several functions available for constructing empty regions, testing to see if a region is empty, and removing instructions from specified queues.

```
0  cnc k m = (int,nxt,ob)
1      where
2      int = do { ((a,b),c) <- initial m
3               ; unit ((a,c),take k (repeat b),emptyRn rs)
4               }
5
6      nxt b (rf,ps,r) =
7              do { choice <- perm rs k
8                 ; let psk = zip ps choice
9                 ; let r' = if isEmptyRn r then fillout rs b else r
10                ; foldM issue (rf , [], r') psk
11                }
12
13     ob ((rf,p'),ps,r) =
14                   let ss = map (\x -> ((rf,x),p')) ps
15                   in if isEmptyRn r &&  all (flushed m) ss
16                      then observe m ((rf,head ps),p')
17                      else r_stalled
18             )
19     issue ((rf,p2), ps',r) (p1,n) =
20       do { let s = ((rf,p1),p2)
21          ; (s',r') <- if stalling m s then match r (next m bubble s)
22                       else case popRn n r of
23                              (Nothing,r') -> match r' (next m bubble s)
24                              (Just x,r')  -> match r' (next m x s)
25          ; let ((rf',p1'),p2') = s'
26          ; unit ((rf',p2'), ps' `union` [p1'],r')
27          }
```

Figure 4.16: cnc: the concurrency transformer

As we can see from line 2 of Figure 4.16, the initial states of the concurrent transition system are built from the underlying transition system's initial states: the first and third elements are returned with `k` copies of the second element. In addition, `cnc` returns an empty region. Lines 6 through 11 define the next-state relation. The variable `choice` is nondeterministically chosen to be a legal assignment of instructions to pipelines. A legal choice is defined as follows: an instruction from queue q can be issued to a pipeline p if $p = q \mod s$, where s represents the number of pipelines. We can also choose not to issue an instruction on that cycle. For example, if we have three pipelines, the next instruction for the first pipeline could come from the first or the fourth queue.

The next state relation is defined with `foldM`:

```
foldM            :: Collect c => (a -> b -> c a) -> a -> [b] -> c a
foldM f a []     = unit a
foldM f a (x:xs) = do {y <- f a x; foldM f y xs}
```

The function `issue` (line 19) issues the choices into their respective pipelines and returns the updated region. On line 20, `issue` makes a state for the underlying system from the available parts. It then issues an instruction for pipeline `n` if possible. A new state `s'` and a reduced region `r'` are returned. The function then de-constructs the state returned by the underlying transition system and adds the pipeline register state into the value that is finally returned.

The observation function (lines 13 to 18) defines the system to be stalled and not flushed unless the region is empty and the pipelines are flushed. In the case that both the region and the pipelines are flushed the definition uses the flushed pipelines to determine the underlying observation.

4.4 Adding the front-end

The last transformer applied in both the architectural and microarchitectural models is `fnt` defined in (Figure 4.17). This function takes a program and inserts instructions from the program into its argument transition system. The type of `fnt` is:

```
fnt p m = (initial m,nxt,observe m)
   where nxt False s = if not (flushed m s) then next m bubble s
                       else let v = readEnv (view m s) pc
                                r = p!!(toInt (v) )
                            in next m r (s)
         nxt True s = next m bubble s
```

Figure 4.17: `fnt`: the instruction memory transformer

```
(Register r, Bubble i, Integral w) =>
  [i] ->
  TS m i    s (Obs (Env r w)) ->
  TS m Bool s (Obs (Env r w))
```

The function returns a transition system with a stall input signal.

Notice that `fnt` is used in four different type instantiations. In WA it is being applied to a predicated RISC architecture, which has the following type:

It is also being applied to a predicated RISC pipeline, a concurrent and predicated RISC architecture and a concurrent and predicated pipeline. We could also apply `fnt` to just a pipeline:

```
fnt p pipe
```

4.5 Executing the specification and model

In this section we demonstrate `oa` and `oma` on the sample program from Chapter 2 which is encoded as a list of regions in the `[Region (Prd_Instr Opcode Reg PReg)]` data-type in Figure 4.18.

4.5.1 Executing the architectural model `oa`

The following is an initial state of `oa`:

```
[
--  Region 0
  [[ [GO (CNT r1 1)]
   , [GO (CNT r2 7)]
   , [GO (CNT pc 1)]
   ]

,
--  Region 1
  ,[ [GO (NEQZ r4 r2), R2P p2 p3 r4]
   , [GO (CNT pc 2), GO (ADDI r3 r2 0)]
   ]

,
--  Region 2
  ,[ [IF (MLT r1 r1 r3) p2]
   , [IF (ADDI r2 r2 (-1)) p2]
   , [IF (CNT pc 1) p2,IF (CNT pc 3) p3]
   , [IF (ADDI r4 r1 0) p3]
   ]

,
--  Region 3
  ,[ [GO (CNT pc 3)]
   , [GO (CNT 6 2),GO (DIVI 5 5 0)]
   ]
  ]
]
```

Figure 4.18: Factorial function encoding

```
((rf,(prf,Nothing)),[3],[[],[],[],[],[],[],[],[],[],[]])
```

where `rf` is the register file ($[\![\texttt{rf pc}]\!] = 0$, $[\![\texttt{rf r1}]\!] = 0$, etc) and `prf` is the predicate register file ($[\![\texttt{prf p0}]\!] = 0$, $[\![\texttt{prf p1}]\!] = 0$, etc). The singleton list `[3]` is `risc`'s non-deterministically chosen stall value. The list of empty lists is the empty region. We format this state as follows:

registers	
pc	0
r0	0
r1	0
r2	0
r3	0
r4	0
r5	0
r6	0
r7	0
⋮	⋮

pred. reg.	
p0	True
p1	True
p2	True
p3	True
p4	True
p5	True
p6	True
p7	True
p8	True
⋮	⋮

region	
1	[]
2	[]
3	[]
4	[]
5	[]
6	[]
7	[]
8	[]
9	[]
10	[]

rgstr
Nothing

On the first transition the first region can be loaded into the system:

registers	
pc	0
r0	0
r1	1
r2	0
r3	0
r4	0
r5	0
r6	0
r7	0
⋮	⋮

pred. reg.	
p0	True
p1	True
p2	True
p3	True
p4	True
p5	True
p6	True
p7	True
p8	True
⋮	⋮

region	
1	[]
2	[GO (CNT r2 7)]
3	[GO (CNT pc 1)]
4	[]
5	[]
6	[]
7	[]
8	[]
9	[]
10	[]

rgstr
Nothing

At the next cycle, because the system is not flushed, the input region is ignored. We can choose to issue the instruction in the third queue:

registers	
pc	1
r0	0
r1	1
r2	0
r3	0
r4	0
r5	0
r6	0
r7	0
⋮	⋮

pred. reg.	
p0	True
p1	True
p2	True
p3	True
p4	True
p5	True
p6	True
p7	True
p8	True
⋮	⋮

region	
1	[]
2	[GO (CNT 2 7)]
3	[]
4	[]
5	[]
6	[]
7	[]
8	[]
9	[]
10	[]

rgstr
Nothing

We can then issue the second instruction on the following cycle:

registers	
pc	1
r0	0
r1	1
r2	7
r3	0
r4	0
r5	0
r6	0
r7	0
⋮	⋮

pred. reg.	
p0	True
p1	True
p2	True
p3	True
p4	True
p5	True
p6	True
p7	True
p8	True
⋮	⋮

region	
1	[]
2	[]
3	[]
4	[]
5	[]
6	[]
7	[]
8	[]
9	[]
10	[]

rgstr
Nothing

Now the program counter (pc) is set to 1 and the system is flushed. Therefore, on the next cycle, the second region can be loaded:

registers	
pc	1
r0	0
r1	1
r2	7
r3	0
r4	1
r5	0
r6	0
r7	0
⋮	⋮

pred. reg.	
p0	True
p1	True
p2	True
p3	True
p4	True
p5	True
p6	True
p7	True
p8	True
⋮	⋮

region	
1	[R2P 2 3 4]
2	[GO (CNT 0 2),GO (ADDI 3 2 0)]
3	[]
4	[]
5	[]
6	[]
7	[]
8	[]
9	[]
10	[]

rgstr
Nothing

4.5.2 Executing the microarchitectural model oma

The implementation model, oma can be analogously executed on the same program. Its initial state is:

registers	
pc	0
r0	0
r1	0
r2	0
r3	0
r4	0
r5	0
r6	0
r7	0
⋮	⋮

pred. reg.	
p0	True
p1	True
p2	True
p3	True
p4	True
p5	True
p6	True
p7	True
p8	True
⋮	⋮

region	
1	[]
2	[]
3	[]
4	[]
5	[]
6	[]
7	[]
8	[]
9	[]
10	[]

pipeline one	
nw	([],[],GO BUBBLE,[],[])
rd	([],[],GO BUBBLE,[],[])
wb	([],[],GO BUBBLE,[],[])

pipeline two	
nw	([],[],GO BUBBLE,[],[])
rd	([],[],GO BUBBLE,[],[])
wb	([],[],GO BUBBLE,[],[])

pipeline three	
nw	([],[],GO BUBBLE,[],[])
rd	([],[],GO BUBBLE,[],[])
wb	([],[],GO BUBBLE,[],[])

After the first transition, the entire first region can be loaded and issued into the three pipelines:

registers	
pc	0
r0	0
r1	0
r2	0
r3	0
r4	0
r5	0
r6	0
r7	0
⋮	⋮

pred. reg.	
p0	True
p1	True
p2	True
p3	True
p4	True
p5	True
p6	True
p7	True
p8	True
⋮	⋮

region	
1	[]
2	[]
3	[]
4	[]
5	[]
6	[]
7	[]
8	[]
9	[]
10	[]

pipeline one	
nw	[(r1,Just 1)],[],GO CNT,[(Nothing,Just 1)],[])
rd	([],[],GO BUBBLE,[],[])
wb	([],[],GO BUBBLE,[],[])

pipeline two	
nw	([(r2,Just 7)],[],GO CNT,[(Nothing,Just 7)],[])
rd	([],[],GO BUBBLE,[],[])
wb	([],[],GO BUBBLE,[],[])

pipeline three	
nw	([(pc,Just 1)],[],GO CNT,[(Nothing,Just 1)],[])
rd	([],[],GO BUBBLE,[],[])
wb	([],[],GO BUBBLE,[],[])

We have now issued all of the instructions in the region. On the next cycle we concurrently run each of the pipelines. Assuming that all of the pipelines make progress, the three transactions can progress to the read stage in their respective pipelines:

registers	
pc	0
r0	0
r1	0
r2	0
r3	0
r4	0
r5	0
r6	0
r7	0
⋮	⋮

pred. reg.	
p0	True
p1	True
p2	True
p3	True
p4	True
p5	True
p6	True
p7	True
p8	True
⋮	⋮

region	
1	[]
2	[]
3	[]
4	[]
5	[]
6	[]
7	[]
8	[]
9	[]
10	[]

pipeline one	
nw	([],[],GO BUBBLE,[],[])
rd	[(r1,Just 1)],[],GO CNT,[(Nothing,Just 1)],[])
wb	([],[],GO BUBBLE,[],[])

pipeline two	
nw	([],[],GO BUBBLE,[],[])
rd	([(r2,Just 7)],[],GO CNT,[(Nothing,Just 7)],[])
wb	([],[],GO BUBBLE,[],[])

pipeline three	
nw	([],[],GO BUBBLE,[],[])
rd	([(pc,Just 1)],[],GO CNT,[(Nothing,Just 1)],[])
wb	([],[],GO BUBBLE,[],[])

During the next transition the transactions progress to the writeback stage of their pipelines:

registers	
pc	0
r0	0
r1	0
r2	0
r3	0
r4	0
r5	0
r6	0
r7	0
⋮	⋮

pred. reg.	
p0	True
p1	True
p2	True
p3	True
p4	True
p5	True
p6	True
p7	True
p8	True
⋮	⋮

region	
1	[]
2	[]
3	[]
4	[]
5	[]
6	[]
7	[]
8	[]
9	[]
10	[]

pipeline one	
nw	([],[],GO BUBBLE,[],[])
rd	([],[],GO BUBBLE,[],[])
wb	([(r1,Just 1)],[],GO CNT,[(Nothing,Just 1)],[])

pipeline two	
nw	([],[],GO BUBBLE,[],[])
rd	([],[],GO BUBBLE,[],[])
wb	([(r1,Just 7)],[],GO CNT,[(Nothing,Just 7)],[])

pipeline three	
nw	([],[],GO BUBBLE,[],[])
rd	([],[],GO BUBBLE,[],[])
wb	([(pc,Just 1)],[],GO CNT,[(Nothing,Just 1)],[])

Finally, on the next cycle, the machine writes the results embedded in the transactions of the three writeback stages:

registers	
pc	1
r0	0
r1	1
r2	7
r3	0
r4	0
r5	0
r6	0
r7	0
⋮	⋮

pred. reg.	
p0	True
p1	True
p2	True
p3	True
p4	True
p5	True
p6	True
p7	True
p8	True
⋮	⋮

region	
1	[]
2	[]
3	[]
4	[]
5	[]
6	[]
7	[]
8	[]
9	[]
10	[]

pipeline one	
nw	([],[],GO BUBBLE,[],[])
rd	([],[],GO BUBBLE,[],[])
wb	([],[],GO BUBBLE,[],[])

pipeline two	
nw	([],[],GO BUBBLE,[],[])
rd	([],[],GO BUBBLE,[],[])
wb	([],[],GO BUBBLE,[],[])

pipeline three	
nw	([],[],GO BUBBLE,[],[])
rd	([],[],GO BUBBLE,[],[])
wb	([],[],GO BUBBLE,[],[])

At this point we have completely executed the first region. On the next cycle, we can load the region pointed to by pc:

registers	
pc	1
r0	0
r1	1
r2	7
r3	0
r4	0
r5	0
r6	0
r7	0
⋮	⋮

pred. reg.	
p0	True
p1	True
p2	True
p3	True
p4	True
p5	True
p6	True
p7	True
p8	True
⋮	⋮

region	
1	[R2P p2 p3 r4]
2	[ADDI r3 r2 0]
3	[]
4	[]
5	[]
6	[]
7	[]
8	[]
9	[]
10	[]

pipeline one	
nw	([(r4,Nothing)],[],GO NEZ,[(Jt r2,Nothing)],[])
rd	([],[],GO BUBBLE,[],[])
wb	([],[],GO BUBBLE,[],[])

pipeline two	
nw	([(pc,Nothing)],[],GO CNT,[(Nothing,Just 2)],[])
rd	([],[],GO BUBBLE,[],[])
wb	([],[],GO BUBBLE,[],[])

pipeline three	
nw	([],[],GO BUBBLE,[],[])
rd	([],[],GO BUBBLE,[],[])
wb	([],[],GO BUBBLE,[],[])

4.6 Summary

In this chapter we have constructed several ISA specifications and microarchitectural designs. The key concept in this chapter is that of a transition system transformer. When represented as the composition of these transformers, the features of a specification and model can be expressed in isolation of other features. For example, the specification of predication is expressed in an orthogonal manner from the specification of concurrency or instruction fetching. Qualified types have played an important role in this method of independently modeling extensions. They have allowed us to encode the expectations of the argument transition system into the type of the transformer. For example, the predication transformer `prd` requires that its argument support the notion of bubbles, so that it can flush its argument transition system in the event of an `R2P` instruction—this expectation is encoded in the predicate `Bubble` which occurs in the function's type.

Another aspect of this chapter is that we have been able to re-use the relatively standard definitions of `risc` and `pipe` in the specifications of OA and WA. For quality it is typically good to build designs using well understood building blocks. What's more: this will be advantageous later when we explore the decomposition of the proof of `wma`'s correctness.

Chapter 5

Proof with transformers

In the previous chapter we demonstrated how we can model architectures and microarchitectures with transformers. We now explore how to use the extra structure in these transformer-based microarchitectural models when formally verifying them against their transformer-based specifications. In this chapter we develop a transformer-based strategy that facilitates the decomposition and simplification of correctness proofs.

The chapter is organized as follows. We first introduce some notation and review mathematical concepts. We develop λ^M, a formal language for the expression of transition systems and transformers. We provide a basic rule that allows for proof decomposition on transformers. Then, using the theory of Parametricity [54, 66] for λ^M, we develop a strategy for proof simplification.

5.1 Notation and mathematical preliminaries

We use $\rightarrow$ and $\times$ over sets in the traditional manner:

Definition 8 ($\times$, $\rightarrow$)

$$\begin{aligned} A \times B &\triangleq \{(a,b) | a \in A \wedge b \in B\} \\ A \rightarrow B &\triangleq \{f | f \subseteq A \times B \wedge \forall x \in A. \exists! y \in B. (x,y) \in f\} \end{aligned}$$

$A \times B$ is the product space of A and B. $A \rightarrow B$ is the function space from A to B.

Later in the chapter, we will need to specify domains and ranges relative to a relation such that the relation appears to be a total relation, a total function, or a one-to-one function. We achieve this with the following higher-order relations:

Definition 9 (total, total_func, iso)

$$\begin{aligned}(A,B) \in \mathsf{total}\ R &\triangleq \forall a \in A.\ \exists b \in B.\ (a,b) \in R \\ (A,B) \in \mathsf{total\_func}\ R &\triangleq \forall a \in A.\ \exists! b \in B.\ (a,b) \in R \\ (A,B) \in \mathsf{iso}\ R &\triangleq (A,B) \in \mathsf{total}\ R \wedge (B,A) \in \mathsf{total}\ R^{-1}\end{aligned}$$

These higher-order relations have type: $Set(A \times B) \rightarrow Set((Set\ A) \times (Set\ B))$. In essence, the relations relate domains to ranges such that they have some property with respect to R. In the case of total, the property is that R appears total under the domain and range. For example, consider total Id, which relates A to B only if A is a subset of B. The relation total_func is identical to total except for the additional constraint that an A and B must be chosen such that R appears to be a function. iso restricts A and B such that the relation is both total and surjective. As an example: iso Id = Id.

We use the notation $(R \leftarrow Q)$ to represent the lifting of relations on transformers:

Definition 10 ($\leftarrow$)

$$(f,g) \in (R \leftarrow Q) \triangleq \forall a, b.\ (a,b) \in Q \Rightarrow (f\ a, g\ b) \in R$$

We use the backwards arrow to simplify composition notationally. When we apply ($\leftarrow$) to relations such as SIM, MAP, BISIM or FP we get new relations that range over transformers. For example: $(f,g) \in (\mathsf{SIM} \leftarrow \mathsf{SIM})$ is true when two machines (x,y) are in SIM and $(f(x), g(y)) \in \mathsf{SIM}$. Figure 5.1 portrays a lattice of implications between lifted relations constructed from SIM, MAP, and FP. It demonstrates that $(\mathsf{FP} \leftarrow \mathsf{MAP})$ is the most general relation, while $(\mathsf{MAP} \leftarrow \mathsf{FP})$ is the least. We can also see that $(\mathsf{MAP} \leftarrow \mathsf{FP}) \subset (\mathsf{FP} \leftarrow \mathsf{SIM})$—meaning that if we have proved $(\mathsf{SIM} \leftarrow \mathsf{FP})$ we have also proved $(\mathsf{FP} \leftarrow \mathsf{SIM})$. This figure assumes that transition systems are all initially flushed.

Definition 11 (monotonicity) *A function $f : A \rightarrow B$ is* monotonic *with respect to R and Q if for all $(a,b) \in Q$, $(f\ a, f\ b) \in R$.*

This is a slight generalization of the familiar definition of monotonicity. Normally we would not use two relations in the definition. Notice that, using the lifting notation, our definition of monotonicity can be rephrased as: A function $f : A \rightarrow B$ is *monotonic* with respect to R and Q if $(f,f) \in (Q \leftarrow R)$.

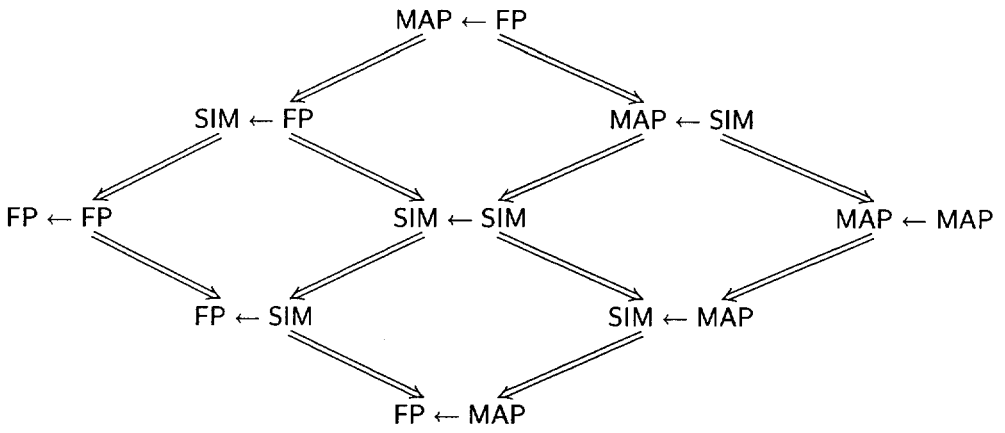


Figure 5.1: Implications between lifted correctness criteria

5.2 λ^M: A language for expressing transition systems

Thus far we have expressed transformers in an unspecified functional programming language notation. We will now be more precise as to the semantics of this language by defining a core language λ^M. The purpose for delving into this level of detail is that, with the language formally defined, we gain the use of Parametricity.

λ^M is minimal and verbose: it is intended as a simple language for study. We conjecture that a compiler could be written that translates the code from Chapters 3 and 4 into terms of λ^M by removing syntactic sugar, compiling away type-classes and performing type inference. Like System F [23], λ^M requires explicit type annotations. Like Haskell or ML, it requires Hindley-Milner types.

We assume that all recursion is bounded. That is, we assume that all recursion can be eliminated by a static number of unfoldings—in practice ten to twenty unfoldings are likely to be more than sufficient. Recursion is used for notational convenience, not computational necessity. This matches the domain in which it is being used—hardware must, in the end, be described with a finite amount of state.

$$
\begin{array}{rcll}
x, y, z & \in & \mathit{Vars} & \\
S & \in & \mathit{TypeSchemes} & ::= \quad \dot{\forall}(\lambda x_1.\dot{\forall}(\lambda x_2 \ldots x_n.T)\ldots) \\
T & \in & \mathit{Types} & ::= \quad x \mid C\ T_1 \ldots T_n \mid T_1 \dot{\rightarrow} T_2 \mid T_1 \dot{\times} T_2 \\
T' & \in & \mathit{GrTypes} & ::= \quad C\ T'_1 \ldots T'_n \mid T'_1 \dot{\rightarrow} T'_2 \mid T'_1 \dot{\times} T'_2 \\
C & \in & \mathit{TypeConstants} & ::= \quad \texttt{FSet} \mid \texttt{Int} \mid \ldots \\
t & \in & \mathit{Terms} & ::= \quad x \mid c \mid \lambda x : T.t \mid \Lambda x.t \mid t_1\ t_2 \mid t_T \mid (t_1, t_2) \\
c & \in & \mathit{Constants} & ::= \quad \texttt{fst} \mid \texttt{snd} \mid \ldots
\end{array}
$$

Figure 5.2: λ^M syntax

5.2.1 Syntax

Figure 5.2 displays the syntax for λ^M types and terms. The letters x, y, and z represent type- and term-variables. To simplify the semantics, we distinguish between type expressions with variables, and *ground type expressions*, which are type expressions without variables.

We represent quantified types with constructors. The familiar type $\forall x.x$ is treated as shorthand for $\dot{\forall}(\lambda x.x)$, where $\dot{\forall}$ is a type constructor and $\lambda x.x$ is a function of the type syntax level from ground types ($\mathit{GrTypes}$) to Types.

Figure 5.2 presents the syntax for λ^M. The form $\lambda x : T.t$ represents abstraction over terms. $\Lambda x.t$ is abstraction over types. The form $(t_1\ t_2)$ denotes the application of term t_1 to t_2. t_T is the applicative form of terms to types.

5.2.2 Types

λ^M borrows system F's type system (Figure 5.3) as described in [66]. A term is considered to be in λ^M if it is well typed. A type assertion is of the form

$$\Delta, \Gamma \vdash t : T$$

Δ is a set of type variables, and Γ is an environment from variables to types. We expect that each free variable in t will be in the domain of Γ. Each free type variable in T should appear in Δ.

$$\frac{\Delta, \Gamma[x \mapsto T_1] \vdash u : T_2}{\Delta, \Gamma \vdash \lambda x : T_1.u : T_1 \dot{\to} T_2} \qquad \frac{\Delta[x], \Gamma \vdash u : T}{\Delta, \Gamma \vdash \Lambda x.u : \dot{\forall}(\lambda x.T)}$$

$$\Delta, \Gamma[x \mapsto T] \vdash x : T \qquad \frac{\Delta, \Gamma \vdash t_1 : T_1 \dot{\to} T_2 \quad \Delta, \Gamma \vdash t_2 : T_1}{\Delta, \Gamma \vdash t_1\ t_2 : T_2}$$

$$\frac{\Delta, \Gamma \vdash t : \dot{\forall}(\lambda x.T_1)}{\Delta, \Gamma \vdash t_{T_2} : T_1[x \mapsto T_2]} \qquad \Delta, \Gamma \vdash c : \sigma_{\mathrm{T}}\ c$$

$$\frac{\Delta, \Gamma \vdash t_1 : T_1 \quad \Delta, \Gamma \vdash t_2 : T_2}{\Delta, \Gamma \vdash (t_1, t_2) : T_1 \dot{\times} T_2}$$

Figure 5.3: Type system of λ^M

5.2.3 Semantics

Because λ^M is intended as the target for a language with only bounded recursion, simple sets suffice as the semantic interpretation. For example, we can represent any function of type `Bool`$\dot{\to}$`Bool` as a subset of the set of pairs:

$$\{(0,0),(0,1),(1,0),(1,1)\}$$

That is, we do not neet to provide an extra element to represent functions that do not terminate. Following the style of Mitchell and Meyer [53], the semantics of λ^M are defined in terms of environments for the typed and kinded constants. σ_{K} provides the meaning for type constants, and τ_{K} gives the kind for each type constant. At the term level, σ_{T} provides the meaning for each term constant, and τ_{T} the types. For example, we will expect the following equations to be true:

$$\begin{aligned}
\tau_{\mathrm{K}}\ \texttt{Bool} &= * \\
\sigma_{\mathrm{K}}\ \texttt{Bool} &= \{1,0\} \\
\tau_{\mathrm{T}}\ \texttt{and} &= \texttt{Bool}\dot{\to}\texttt{Bool}\dot{\to}\texttt{Bool} \\
\sigma_{\mathrm{T}}\ \texttt{and} &= \lambda x.\lambda y.x \wedge y
\end{aligned}$$

We provide the semantics of λ^M in a frame-based form, as described by Meyer and Bruce [15]. That is, we provide a frame $(T, S, \dot{\to}, \dot{\forall}, \Phi, \Psi)$. From this frame, Meyer and Bruce

$$
\begin{array}{lcl}
[\![x]\!]_\psi & \triangleq & \psi\ x \\
[\![C\ T_1 \ldots T_n]\!]_\psi & \triangleq & (\sigma_{\mathtt{K}}\ C)\ [\![T_1]\!]_\psi \ldots [\![T_n]\!]_\psi \\
[\![T_1 \dot{\to} T_2]\!]_\psi & \triangleq & [\![T_1]\!]_\psi \to [\![T_2]\!]_\psi \\
[\![T_1 \dot{\times} T_2]\!]_\psi & \triangleq & [\![T_1]\!]_\psi \times [\![T_2]\!]_\psi \\
[\![\dot{\forall}(\lambda x.T)]\!]_\psi & \triangleq & \lambda A : GSet.\ [\![T]\!]_{\psi[x \mapsto A]}
\end{array}
$$

Figure 5.4: Type semantics of λ^M

$$
\begin{array}{lcl}
[\![x]\!]_{\psi,\rho} & \triangleq & \rho\ x \\
[\![c]\!]_{\psi,\rho} & \triangleq & \sigma_{\mathtt{T}}\ c \\
[\![\lambda x : U.v]\!]_{\psi,\rho} & \triangleq & \lambda a : [\![T]\!]_\psi .\ [\![v]\!]_{\psi,\rho[x \mapsto a]} \\
[\![t\ u]\!]_{\psi,\rho} & \triangleq & [\![t]\!]_{\psi,\rho}\ [\![u]\!]_{\psi,\rho} \\
[\![\Lambda x.\ t]\!]_{\psi,\rho} & \triangleq & \lambda A : GSet.\ [\![t]\!]_{\psi[x \mapsto A],\rho} \\
[\![t_T]\!]_{\psi,\rho} & \triangleq & [\![t]\!]_{\psi,\rho}\ [\![T]\!]_\psi
\end{array}
$$

Figure 5.5: Term semantics of λ^M

construct $[\![\cdot]\!]$ for terms and types defined in Figures 5.4 and 5.5. Both Φ and Ψ are in this case the identity function.

The function S (Figure 5.6) takes a type expression and returns the set of elements that represent that type. For example: $S\ \mathtt{Bool} = \{\mathtt{True}, \mathtt{False}\}$. $GSet$ is defined as the meaning of all possible monomorphic types:

$$GSet \triangleq \{[\![t]\!] | t \in GrTypes\}$$

The meaning of a polymorphic term $t : \dot{\forall}(\lambda x.T)$ is represented as a environment from

$$
\begin{array}{rcl}
S(T_1 \dot{\to} T_2) & \triangleq & S\ T_1 \to S\ T_2 \\
S(C\ T_1 \ldots T_n) & \triangleq & (\sigma_k\ C)\ S(T_1) \ldots S(T_n) \\
S(\dot{\forall} f) & \triangleq & \{g | g : T' \to \bigcup_{t \in T} D_{(f\ t)} \wedge \forall t \in T'.\ g\ t \in D_{(f\ t)}\}
\end{array}
$$

Figure 5.6: S: a mapping from λ^M-types to sets of λ^M-expressions

$$
\begin{array}{rcl}
(a,b) \in \mathsf{Rel}_\zeta\ x & \triangleq & \zeta\ x \\
(a,b) \in \mathsf{Rel}_\zeta\ c & \triangleq & \kappa\ c \\
(a,b) \in \mathsf{Rel}_\zeta\ (\dot{\forall}(\lambda x.\ T)) & \triangleq & \forall R.\ (a,b) \in \mathsf{Rel}_{\zeta[x \mapsto R]}\ T \\
(a,b) \in \mathsf{Rel}_\zeta\ (T_1 \dot{\to} T_2) & \triangleq & (a,b) \in (\mathsf{Rel}_\zeta\ T_2 \leftarrow \mathsf{Rel}_\zeta\ T_1) \\
(a,b) \in \mathsf{Rel}_\zeta\ (T_1 \dot{\times} T_2) & \triangleq & (a,b) \in (\mathsf{Rel}_\zeta\ T_1 \times \mathsf{Rel}_\zeta\ T_2)
\end{array}
$$

Figure 5.7: Rel: an alternative semantics for types

GSet to the meaning of terms. For example,

$$
\begin{array}{rcl}
[\![(\Lambda x.\ \lambda y : x.\ \lambda z : x.y)]\!] & = & \lambda A : GSet.\ [\![(\lambda y : x.\ \lambda z : x.y)]\!]_{\emptyset,[x \mapsto A]} \\
& = & \lambda A : GSet.\ \lambda a_1 : [\![x]\!]_{[x \mapsto A]}.\ [\![\lambda z : x.y]\!]_{[y \mapsto a_1],[x \mapsto A]} \\
& = & \lambda A : GSet.\ \lambda a_1 : A.\ \lambda a_2 : A.\ [\![y]\!]_{[y \mapsto a_1, z \mapsto a_2],[x \mapsto A]} \\
& = & \lambda A : GSet.\ \lambda a_1 : A.\ \lambda a_2 : A.\ a_1
\end{array}
$$

This is a significant departure from the standard semantics for System F. This restriction essentially states that polymorphism only ranges over ground types. The form $\lambda A : B.t$ is a notational shorthand for the set of pairs $\{(A,t) | A \in B\}$. We use $[\![\cdot]\!]$ as an abbreviation for $[\![\cdot]\!]_\emptyset$ and $[\![\cdot]\!]_{\emptyset,\emptyset}$.

5.3 Parametricity for λ^M

In essence, Parametricity [54, 66] states that for each type there is a theorem that holds for any expression of that type. In other words, for every term with type $t : T$ we know that there is a statement constructed from T that is true of t—regardless of t's definition. The relation Rel (see Figure 5.7) builds this statement.

Now for a more formal explanation. A constant c is said to be *parametric* if $(\sigma_\mathrm{T}\ c, \sigma_\mathrm{T}\ c) \in \mathsf{Rel}\ T$. Parametricity states that if every constant is parametric, then for any term $t : T$, $([\![t]\!], [\![t]\!]) \in \mathsf{Rel}\ T$. Rel T is sometimes called T's *free theorem*. Note that Rel uses ζ as an environment from variables to relations, which grows in the recursive applications of Rel over quantified types. We will use Rel as an abbreviation for $\mathsf{Rel}_\emptyset$. The environment κ is defined such that a (possibly higher-order) relation is bound for each type-constant in the language. For example, κ `Int` = Id and κ `FSet` could be defined as iso.

Let us see a few examples of free theorems. The relation that compares two sets A and B which are denoted by expressions of type `FSet Int` is

$$\begin{aligned}
(A,B) \in \mathsf{Rel}\ (\texttt{FSet Int}) &\Leftrightarrow (A,B) \in (\mathsf{Rel}\ \texttt{FSet})\ (\mathsf{Rel}\ \texttt{Int}) \\
&\Leftrightarrow (A,B) \in \mathsf{iso}\ (\mathsf{Rel}\ \texttt{Int}) \\
&\Leftrightarrow (A,B) \in \mathsf{iso}\ \mathsf{Id} \\
&\Leftrightarrow (A,B) \in \mathsf{total}\ \mathsf{Id} \wedge (B,A) \in \mathsf{total}\ \mathsf{Id}^{-1} \\
&\Leftrightarrow \forall a \in A.\ \exists b \in B.\ (a,b) \in \mathsf{Id} \wedge \forall b \in B.\ \exists a \in A.\ (b,a) \in \mathsf{Id} \\
&\Leftrightarrow (\forall a \in A.\ \exists b \in B.\ a = b) \wedge (\forall b \in B.\ \exists a \in A.\ b = a) \\
&\Leftrightarrow A = B
\end{aligned}$$

So the free theorem of any term t with type `FSet Int` is that $[\![t]\!] = [\![t]\!]$. This is not a ground breaking theorem, but it's free.

For a more interesting free theorem, recall that in Chapter 4 we wrote transformers with type: $\forall x.\ \texttt{TS FSet}\ x\ T_1\ T_2 \rightarrow \texttt{TS FSet}\ (F\ x)\ T_3\ T_4$, where F is any function at the syntactic type level. Let us take a specific example. Suppose that we have an encoding of a transformer in λ^M:

$$f :: \forall x.\ \texttt{TS FSet}\ x\ T_1\ T_2 \rightarrow \texttt{TS FSet}\ (x \dot{\times} x)\ T_3\ T_4$$

What is the theorem associated with this type? Recall that Rel constructs a relation from a type that compares two elements. In the case of TS, Rel (TS $T_1\ T_2\ T_3\ T_4$) is constructed to relate two transition systems:

$$\begin{array}{c}
\texttt{TS}\ T_1\ T_2\ T_3\ T_4 \\
\uparrow \\
\mathsf{Rel}\ (\texttt{TS}\ T_1\ T_2\ T_3\ T_4) \\
\downarrow \\
\texttt{TS}\ T_1\ T_5\ T_3\ T_4
\end{array}$$

This picture is intended to communicate that Rel (TS $T_1\ T_2\ T_3\ T_4$) is a relation that holds between elements in TS $T_1\ T_2\ T_3\ T_4$ and TS $T_1\ T_5\ T_3\ T_4$. In other words,

$$\mathsf{Rel}\ (\texttt{TS}\ T_1\ T_2\ T_3\ T_4) :: \texttt{TS}\ T_1\ T_2\ T_3\ T_4 \rightarrow \texttt{TS}\ T_1\ T_5\ T_3\ T_4 \rightarrow \{0,1\}$$

Let $R_i = \mathsf{Rel}\ T_i$. The picture below demonstrates how the structure of TS is used to construct the relationship $\mathsf{Rel}\ (\mathtt{TS}\ T_1\ T_2\ T_3\ T_4)$:

$$
\begin{array}{lcl}
\mathtt{TS}\ T_1\ T_2\ T_3\ T_4 & = & (\ T_1\ T_2\ ,\ T_3 \dashrightarrow T_2 \dashrightarrow T_1\ T_2\ ,\ T_2 \dashrightarrow T_4\ ) \\
\Updownarrow & & \Updownarrow\ \ \ \Updownarrow\ \ \ \Updownarrow\ \ \ \Updownarrow\ \ \ \Updownarrow\ \ \ \Updownarrow \\
\mathsf{Rel}\ (\mathtt{TS}\ T_1\ T_2\ T_3\ T_4) & = & R_1\ R_2 \wedge R_3 \Rightarrow R_2 \Rightarrow R_1\ R_2 \wedge R_2 \Rightarrow R_4 \\
\Updownarrow & & \Updownarrow\ \ \ \Updownarrow\ \ \ \Updownarrow\ \ \ \Updownarrow\ \ \ \Updownarrow\ \ \ \Updownarrow \\
\mathtt{TS}\ T_1\ T_5\ T_3\ T_4 & = & (\ T_1\ T_5\ ,\ T_3 \dashrightarrow T_5 \dashrightarrow T_1\ T_5\ ,\ T_5 \dashrightarrow T_4\ )
\end{array}
$$

Notice how the structure of TS naturally causes Rel to construct the relation with three conjuncted clauses:

$$
\begin{array}{lcl}
\mathtt{TS}\ T_1\ T_2\ T_3\ T_4 & = & (\ \boxed{T_1\ T_2}\ ,\ \boxed{T_3 \dashrightarrow T_2 \dashrightarrow T_1\ T_2}\ ,\ \boxed{T_2 \dashrightarrow T_4}\ ) \\
\Updownarrow & & \Updownarrow\ \ \ \Updownarrow\ \ \ \Updownarrow\ \ \ \Updownarrow\ \ \ \Updownarrow \\
\mathsf{Rel}\ (\mathtt{TS}\ T_1\ T_2\ T_3\ T_4) & = & \boxed{R_1\ R_2} \wedge \boxed{R_3 \Rightarrow R_2 \Rightarrow R_1\ R_2} \wedge \boxed{R_2 \Rightarrow R_4} \\
\Updownarrow & & \Updownarrow\ \ \ \Updownarrow\ \ \ \Updownarrow\ \ \ \Updownarrow\ \ \ \Updownarrow \\
\mathtt{TS}\ T_1\ T_5\ T_3\ T_4 & = & (\ \boxed{T_1\ T_5}\ ,\ \boxed{T_3 \dashrightarrow T_5 \dashrightarrow T_1\ T_5}\ ,\ \boxed{T_5 \dashrightarrow T_4}\ )
\end{array}
$$

The first clause compares the first projection of the triples (`initial`):

$$
\boxed{\begin{array}{c} T_1\ T_2 \\ \Updownarrow \\ R_1\ R_2 \\ \Downarrow \\ T_1\ T_5 \end{array}}
$$

That is, if `initial` $u : T_1\ T_2$ and `initial` $v : T_1\ T_5$ then `initial` u can be compared to `initial` v with the relation $(R_1\ R_2)$: $(\texttt{initial}\ u, \texttt{initial}\ v) \in (R_1\ R_2)$. For example, if $R_1 = \mathsf{iso}$, then $R_1\ R_2$ equals

$$\forall a \in \texttt{initial}\ u.\ \exists b \in \texttt{initial}\ v.(a, b) \in R_2$$

$$\wedge$$

$$\forall b \in \texttt{initial}\ v.\ \exists a \in \texttt{initial}\ u.(a, b) \in R_2$$

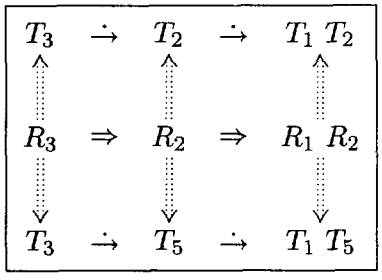
This precisely matches BISIM.A, which was defined in Chapter 3 as:

$$\forall a \in \texttt{initial}\ u.\ \exists b \in \texttt{initial}\ v.(a,b) \in R$$

and

$$\forall b \in \texttt{initial}\ v.\ \exists a \in \texttt{initial}\ u.(a,b) \in R$$

The second clause can be visually represented as:



That is, this clause constructs a relation that compares the second projection of the triples. To compare two elements f and g with R_2, R_3, and $(R_1 R_2)$:

$$f :: T_3 \to T_2 \to T_1\ T_2$$

$$g :: T_3 \to T_5 \to T_5\ T_2$$

$$R_2 :: T_3 \to T_5 \to \{0,1\}$$

$$R_3 :: T_3 \to T_3 \to \{0,1\}$$

$$R_1\ R_2 :: T_1\ T_2 \to T_1\ T_5 \to \{0,1\}$$

we do the following: assume that $(i,j) \in R_3$; assume that $(s,q) \in R_2$; and then check that $(f\ i\ s, g\ j\ q) \in R_1\ R_2$. For example, if R_3 is equality, $R_1 = \mathsf{iso}$, $\texttt{next}\ u :: T_3 \dot{\to} T_2 \dot{\to} T_1\ T_2$, and $\texttt{next}\ v :: T_3 \dot{\to} T_5 \dot{\to} T_1\ T_5$, then this is equivalent to:

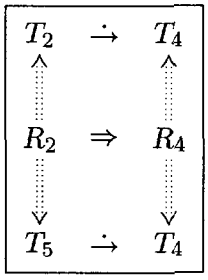
$$\forall a, a', b, i.\ [R_2(a,b) \wedge a' \in \texttt{next}\ u\ i\ a] \Rightarrow [\exists b'.\ b' \in \texttt{next}\ v\ i\ b \wedge R_2(a',b')]$$

$$\wedge$$

$$\forall a, b, a', i.\ [R_2(a,b) \wedge b' \in \texttt{next}\ v\ i\ b] \Rightarrow [\exists a'.\ a' \in \texttt{next}\ u\ i\ a \wedge R_2(a',b')]$$

This matches BISIM.B.

The final clause is:



This clause constructs a relation that compares the third projection. If R_4 is equality, `observe` $u :: T_2 \dot{\to} T_4$, and `observe` $v :: T_5 \dot{\to} T_4$ then this is equivalent to:

$$\forall a, b.\ (a, b) \in R_2 \Rightarrow \texttt{observe}\ u\ a = \texttt{observe}\ v\ b$$

This matches BISIM's third clause:

$$\forall a, b.\ (a, b) \in R \Rightarrow \texttt{observe}\ u\ a = \texttt{observe}\ v\ b$$

This connection between Rel and BISIM provides us with the following result:

Proposition 3 *Given* $u :: \dot{\forall}(\lambda x.\ \texttt{TS FSet}\ (F\ x)\ T_2\ T_3)$, $v :: \dot{\forall}(\lambda x.\ \texttt{TS FSet}\ (F\ x)\ T_2\ T_3)$, *and* $\kappa\ \texttt{FSet} = \mathsf{iso}$,

$$(u, v) \in \mathsf{BISIM}_R \text{ equals } (u, v) \in \mathsf{Rel}_{[x \mapsto R]}\ (\texttt{TS FSet}\ (F\ x)\ T_2\ T_3)$$

With this proposition, we are now ready to see what the free theorem is for f, where f has type:

$$\dot{\forall}(\lambda x.\ \texttt{TS FSet}\ x\ T_1\ T_2 \dot{\to} \texttt{TS FSet}\ (x \dot{\times} x)\ T_3\ T_4)$$

By the Parametricity theorem for λ^M, we know that:

$$([\![f]\!], [\![f]\!]) \in \mathsf{Rel}\ (\dot{\forall}(\lambda x.\ \texttt{TS FSet}\ x\ T_1\ T_2 \dot{\to} \texttt{TS FSet}\ (x \dot{\times} x)\ T_3\ T_4))$$

from this fact we can derive the following:

$$\begin{aligned}
&\Leftrightarrow\ \forall R.([\![f]\!], [\![f]\!]) \in \mathsf{Rel}_{[x \mapsto R]}\ (\texttt{TS FSet}\ x\ T_1\ T_2 \dot{\to} \texttt{TS FSet}\ (x \dot{\times} x)\ T_3\ T_4) \\
&\Leftrightarrow\ \forall R, a, b.\ (a, b) \in \mathsf{Rel}_{[x \mapsto R]}\ (\texttt{TS FSet}\ x\ T_1\ T_2) \\
&\qquad \Rightarrow ([\![f]\!]\ a, [\![f]\!]\ b) \in \mathsf{Rel}_{[x \mapsto R]}\ (\texttt{TS FSet}\ (x \dot{\times} x)\ T_3\ T_4) \\
&\Leftrightarrow\ \forall R, a, b.\ (a, b) \in \mathsf{BISIM}_R \Rightarrow ([\![f]\!]\ a, [\![f]\!]\ b) \in \mathsf{BISIM}_{(R \times R)} \\
&\Rightarrow\ \forall a, b.\ (a, b) \in \mathsf{BISIM} \Rightarrow ([\![f]\!]\ a, [\![f]\!]\ b) \in \mathsf{BISIM} \\
&\Leftrightarrow\ ([\![f]\!], [\![f]\!]) \in (\mathsf{BISIM} \leftarrow \mathsf{BISIM})
\end{aligned}$$

That is, the free theorem states that f is monotonic with respect to BISIM. Remember that this theorem is based completely on f's type. More generally, we can do the same trick for any function, f, with type:

$$\dot{\forall}(\lambda x.\ \texttt{TS FSet}\ x\ T_1\ T_2 \dot{\rightarrow} \texttt{TS}\ M\ (F\ x)\ T_3\ T_4)$$

If $(m, n) \in \mathsf{BISIM}_R$ then $(f\ m, f\ n) \in \mathsf{BISIM}_{(\mathsf{Rel}_{\lfloor x \rightharpoonup R \rfloor}(F\ x))}$. Therefore, we can conclude that any transformer of this general type is monotonic with respect to BISIM.

5.4 Parametricity and Collect

Unfortunately, transformers of type $\dot{\forall}(\lambda x.\texttt{TS FSet}\ x\ T_1\ T_2 \dot{\rightarrow} \texttt{TS FSet}\ (F\ x)\ T_3\ T_4)$ are not necessarily monotonic with respect to SIM. Consider the following example, which inspects how many transitions exist from the set of initial states.

```
f: TS FSet x () Bool -> TS FSet x () Bool
f m = (init m,next m,ob)
   where
   ob = if size (join (map (next m ()) (init m)))== 1 then obs m
        else \x.False
```

The trouble is that the two machines in the relationship SIM might return sets of states with different cardinalities. Consider the following transition systems `m` and `n` :

```
m = (unit 1,\x y -> unit (x+x),\x -> (x % 2) == 0)
n = (unit 1,\x y -> unit (x+x) `union` x,\x -> (x % 2) == 0)
```

In this case, $([\![\texttt{m}]\!], [\![\texttt{n}]\!]) \in \mathsf{SIM}$, but $([\![\texttt{f m}]\!], [\![\texttt{f n}]\!]) \notin \mathsf{SIM}$.

However, with type classes we can restrict the available functions over sets to those of `Collect`. To do this, we must first add `Collect` into λ^M, where $\tau_{\mathrm{K}}\ \texttt{Collect} \triangleq * \rightarrow *$:

$$\begin{aligned}
\tau_{\mathrm{T}}\ \texttt{unit} &\triangleq \dot{\forall}(\lambda x.x \dot{\rightarrow} \texttt{Collect}\ x) \\
\tau_{\mathrm{T}}\ \texttt{join} &\triangleq \dot{\forall}(\lambda x.\texttt{Collect}\ (\texttt{Collect}\ x) \dot{\rightarrow} \texttt{Collect}\ x) \\
\tau_{\mathrm{T}}\ \texttt{union} &\triangleq \dot{\forall}(\lambda x.\texttt{Collect}\ x \dot{\rightarrow} \texttt{Collect}\ x \dot{\rightarrow} \texttt{Collect}\ x) \\
\tau_{\mathrm{T}}\ \texttt{map} &\triangleq \dot{\forall}(\lambda x.\dot{\forall}(\lambda y.(x \dot{\rightarrow} y) \dot{\rightarrow} \texttt{Collect}\ x \dot{\rightarrow} \texttt{Collect}\ y))
\end{aligned}$$

We can then encode the transformers from Chapter 4 that use `Collect` as functions in λ^M with type:

$$(\texttt{Collect}\ T_2, T_3 \dot{\rightarrow} T_2 \dot{\rightarrow} \texttt{Collect}\ T_2, T_2 \dot{\rightarrow} T_4)$$

$$\dot{\rightarrow}$$

$$(\texttt{Collect}\ T_3, T_3 \dot{\rightarrow} T_4 \dot{\rightarrow} \texttt{Collect}\ T_3, T_2 \dot{\rightarrow} T_5)$$

We have a number of interpretations in mind for `Collect`: `One`, `FSet`, etc. Therefore, the semantics of `Collect` are provided in axiomatic form. The axioms guarantee that the constants of `Collect` are parametric:

$$\begin{aligned}
\textbf{A1}: &\quad (\sigma_{\mathrm{T}}\ \texttt{unit}, \sigma_{\mathrm{T}}\ \texttt{unit}) \in \mathsf{Rel}\ (\dot{\forall}(\lambda x.x \dot{\rightarrow} \texttt{Collect}\ x)) \\
\textbf{A2}: &\quad (\sigma_{\mathrm{T}}\ \texttt{join}, \sigma_{\mathrm{T}}\ \texttt{join}) \in \mathsf{Rel}\ (\dot{\forall}(\lambda x.\texttt{Collect}\ (\texttt{Collect}\ x) \dot{\rightarrow} \texttt{Collect}\ x)) \\
\textbf{A3}: &\quad (\sigma_{\mathrm{T}}\ \texttt{union}, \sigma_{\mathrm{T}}\ \texttt{union}) \in \mathsf{Rel}\ (\dot{\forall}(\lambda x.\texttt{Collect}\ x \dot{\rightarrow} \texttt{Collect}\ x \dot{\rightarrow} \texttt{Collect}\ x)) \\
\textbf{A4}: &\quad (\sigma_{\mathrm{T}}\ \texttt{map}, \sigma_{\mathrm{T}}\ \texttt{map}) \in \mathsf{Rel}\ (\dot{\forall}(\lambda x.\dot{\forall}(\lambda y.(x \dot{\rightarrow} y) \dot{\rightarrow} \texttt{Collect}\ x \dot{\rightarrow} \texttt{Collect}\ y)))
\end{aligned}$$

If our interpretation of `Collect` is finite sets, we can define the meaning of the constants as:

$$\begin{aligned}
\sigma_{\mathrm{T}}\ \texttt{unit} &\triangleq \{\cdot\} \\
\sigma_{\mathrm{T}}\ \texttt{join} &\triangleq \bigcup \\
\sigma_{\mathrm{T}}\ \texttt{union} &\triangleq \cup \\
\sigma_{\mathrm{T}}\ \texttt{map} &\triangleq \lambda f.\lambda x.\{f\ x | x \in X\}
\end{aligned}$$

We can implement `Collect` in κ as total. We must, however, verify that the model meets axioms **A1** through **A4**. That is, we need to show that the model is parametric.

Proposition 4 *If* $\sigma_{\mathrm{T}}\ \texttt{unit} = \{\cdot\}$ *and* $\kappa\ \texttt{Collect} = \mathsf{total}$,

$$\textbf{A1}: (\sigma_{\mathrm{T}}\ \texttt{unit}, \sigma_{\mathrm{T}}\ \texttt{unit}) \in \mathsf{Rel}\ (\dot{\forall}(\lambda x.x \dot{\rightarrow} \texttt{Collect}\ x))$$

Proof.

$$
\begin{aligned}
& (\sigma_{\mathrm{T}}\ \mathtt{unit}, \sigma_{\mathrm{T}}\ \mathtt{unit}) \in \mathsf{Rel}\ (\dot{\forall}(\lambda x.x \dot{\rightarrow} \mathtt{Collect}\ x)) \\
\Leftrightarrow\ & (\lambda y.\{y\}, \lambda y.\{y\}) \in \mathsf{Rel}\ (\dot{\forall}(\lambda x.x \dot{\rightarrow} \mathtt{Collect}\ x)) \\
\Leftrightarrow\ & (\lambda y.\{y\}, \lambda y.\{y\}) \in \mathsf{Rel}_{[x \mapsto R]}\ (x \dot{\rightarrow} \mathtt{Collect}\ x) \\
\Leftrightarrow\ & (\lambda y.\{y\}, \lambda y.\{y\}) \in (\mathsf{Rel}_{[x \mapsto R]}\ (\mathtt{Collect}\ x) \leftarrow \mathsf{Rel}_{[x \mapsto R]}\ x) \\
\Leftrightarrow\ & \forall a, b.(a, b) \in \mathsf{Rel}_{[x \mapsto R]}\ x \Rightarrow (\{a\}, \{b\}) \in \mathsf{Rel}_{[x \mapsto R]}\ (\mathtt{Collect}\ x) \\
\Leftrightarrow\ & \forall a, b.(a, b) \in R \Rightarrow (\{a\}, \{b\}) \in \mathsf{total}\ (\mathsf{Rel}_{[x \mapsto R]}\ x) \\
\Leftrightarrow\ & \forall a, b.(a, b) \in R \Rightarrow (\{a\}, \{b\}) \in \mathsf{total}\ R \\
\Leftrightarrow\ & \forall a, b.(a, b) \in R \Rightarrow \forall c \in \{a\}.\ \exists d \in \{b\}.\ (c, d) \in R \\
\Leftrightarrow\ & \mathsf{true}
\end{aligned}
$$

□

Proposition 5 *If* $\sigma_{\mathrm{T}}\ \mathtt{union} = \cup$ *and* $\kappa\ \mathtt{Collect} = \mathsf{total}$,

$$A3 : (\sigma_{\mathrm{T}}\ \mathtt{union}, \sigma_{\mathrm{T}}\ \mathtt{union}) \in \mathsf{Rel}\ ((\dot{\forall}(\lambda x.\mathtt{Collect}\ x \dot{\rightarrow} \mathtt{Collect}\ x \dot{\rightarrow} \mathtt{Collect}\ x)$$

Proposition 6 *If* $\sigma_{\mathrm{T}}\ \mathtt{join} = \bigcup$ *and* $\kappa\ \mathtt{Collect} = \mathsf{total}$,

$$\mathbf{A2} : (\sigma_{\mathrm{T}}\ \mathtt{join}, \sigma_{\mathrm{T}}\ \mathtt{join}) \in \mathsf{Rel}\ (\dot{\forall}(\lambda x.\mathtt{Collect}\ (\mathtt{Collect}\ x) \dot{\rightarrow} \mathtt{Collect}\ x))$$

Proof.

$$
\begin{aligned}
& (\sigma_{\mathrm{T}}\ \texttt{join}, \sigma_{\mathrm{T}}\ \texttt{join}) \in \mathsf{Rel}\ (\dot{\forall}(\lambda x.\texttt{Collect}\ (\texttt{Collect}\ x) \dot{\rightarrow} \texttt{Collect}\ x)) \\
\Leftrightarrow\ & (\bigcup, \bigcup) \in \mathsf{Rel}\ (\dot{\forall}(\lambda x.\texttt{Collect}\ (\texttt{Collect}\ x) \dot{\rightarrow} \texttt{Collect}\ x)) \\
\Leftrightarrow\ & (\bigcup, \bigcup) \in \mathsf{Rel}_{[x \mapsto R]}\ (\texttt{Collect}\ (\texttt{Collect}\ x) \dot{\rightarrow} \texttt{Collect}\ x) \\
\Leftrightarrow\ & (\bigcup, \bigcup) \in (\mathsf{Rel}_{[x \mapsto R]}\ (\texttt{Collect}\ x) \leftarrow \mathsf{Rel}_{[x \mapsto R]}\ (\texttt{Collect}\ (\texttt{Collect}\ x))) \\
\Leftrightarrow\ & \forall A, B.\ [(A, B) \in \mathsf{Rel}_{[x \mapsto R]}\ (\texttt{Collect}\ (\texttt{Collect}\ x)) \\
& \Rightarrow (\bigcup A, \bigcup B) \in \mathsf{Rel}_{[x \mapsto R]}\ (\texttt{Collect}\ x) \\
\Leftrightarrow\ & \forall A, B.\ [(A, B) \in \mathsf{Rel}_{[x \mapsto R]}\ (\texttt{Collect}\ (\texttt{Collect}\ x)) \\
& \Rightarrow (\bigcup A, \bigcup B) \in \mathsf{total}\ (\mathsf{Rel}_{[x \mapsto R]}\ x) \\
\Leftrightarrow\ & \forall A, B.\ [(A, B) \in \mathsf{total}\ (\mathsf{Rel}_{[x \mapsto R]}\ (\texttt{Collect}\ x)) \\
& \Rightarrow (\bigcup A, \bigcup B) \in \mathsf{total}\ (\mathsf{Rel}_{[x \mapsto R]}\ x) \\
\Leftrightarrow\ & \forall A, B.\ [(A, B) \in \mathsf{total}\ (\mathsf{total}\ (\mathsf{Rel}_{[x \mapsto R]}\ x)) \\
& \Rightarrow (\bigcup A, \bigcup B) \in \mathsf{total}\ (\mathsf{Rel}_{[x \mapsto R]}\ x) \\
\Leftrightarrow\ & \forall A, B.\ [(A, B) \in \mathsf{total}\ (\mathsf{total}\ R) \\
& \Rightarrow (\bigcup A, \bigcup B) \in \mathsf{total}\ (\mathsf{Rel}_{[x \mapsto R]}\ x)] \\
\Leftrightarrow\ & \forall A, B.\ [(A, B) \in \mathsf{total}\ (\mathsf{total}\ R) \\
& \Rightarrow (\bigcup A, \bigcup B) \in \mathsf{total}\ R \\
\Leftrightarrow\ & \forall A, B.\ [\forall A' \in A.\ \exists B' \in B.\ (A', B') \in \mathsf{total}\ R \\
& \Rightarrow (\bigcup A, \bigcup B) \in \mathsf{total}\ R \\
\Leftrightarrow\ & \forall A, B.\ [\forall A' \in A.\ \exists B' \in B.\ \forall a \in A'. \exists b \in B'.\ (a, b) \in R \\
& \Rightarrow (\bigcup A, \bigcup b) \in \mathsf{total}\ R \\
\Leftrightarrow\ & \forall A, B.\ [\forall A' \in A.\ \exists B' \in B.\ \forall a \in A'. \exists b \in B'.\ (a, b) \in R \\
& \Rightarrow \forall a \in \bigcup A.\ \exists b \in \bigcup B.\ (x, y) \in R \\
\Leftrightarrow\ & \mathsf{true}
\end{aligned}
$$

□

Proposition 7 *If* $\sigma_{\mathrm{T}}\ \texttt{map} = \lambda f.\lambda A.\{f\ a | a \in A\}$ *and* $\kappa\ \texttt{Collect} = \mathsf{total}$,

$$\mathbf{A4}: (\sigma_{\mathrm{T}}\ \texttt{map}, \sigma_{\mathrm{T}}\ \texttt{map}) \in \mathsf{Rel}\ (\dot{\forall}(\lambda x.\dot{\forall}(\lambda y.(x \dot{\rightarrow} y) \dot{\rightarrow} \texttt{Collect}\ x \dot{\rightarrow} \texttt{Collect}\ y)))$$

If the interpretation of `Collect` is `One`, then we can define the meaning of the constants as:

$$\begin{aligned}
\sigma_{\mathrm{T}}\ \texttt{unit} &\triangleq \mathsf{Id} \\
\sigma_{\mathrm{T}}\ \texttt{join} &\triangleq \mathsf{Id} \\
\sigma_{\mathrm{T}}\ \texttt{union} &\triangleq \lambda a.\ \lambda b.\ a \\
\sigma_{\mathrm{T}}\ \texttt{map} &\triangleq \lambda f.\lambda a.f\ a
\end{aligned}$$

In this case we implement `Collect` in κ as $\kappa\ \texttt{Collect} = \mathsf{Id}$.

Proposition 8 *If* $\sigma_{\mathrm{T}}\ \texttt{unit} = \mathsf{Id}$ *and* $\kappa\ \texttt{Collect} = \mathsf{Id}$,

$$\mathbf{A1}: (\sigma_{\mathrm{T}}\ \texttt{unit}, \sigma_{\mathrm{T}}\ \texttt{unit}) \in \mathsf{Rel}\ (\dot{\forall}(\lambda x.x \dot{\rightarrow} \texttt{Collect}\ x))$$

Proposition 9 *If* $\sigma_{\mathrm{T}}\ \texttt{join} = \mathsf{Id}$ *and* $\kappa\ \texttt{Collect} = \mathsf{Id}$,

$$\mathbf{A2}: (\sigma_{\mathrm{T}}\ \texttt{join}, \sigma_{\mathrm{T}}\ \texttt{join}) \in \mathsf{Rel}\ (\dot{\forall}(\lambda x.\texttt{Collect}\ (\texttt{Collect}\ x) \dot{\rightarrow} \texttt{Collect}\ x))$$

Proof.

$$\begin{aligned}
& (\sigma_{\mathrm{T}}\ \texttt{join}, \sigma_{\mathrm{T}}\ \texttt{join}) \in \mathsf{Rel}\ (\dot{\forall}(\lambda x.\texttt{Collect}\ (\texttt{Collect}\ x) \dot{\rightarrow} \texttt{Collect}\ x)) \\
\Leftrightarrow\ & (\mathsf{Id}, \mathsf{Id}) \in \mathsf{Rel}\ (\dot{\forall}(\lambda x.\texttt{Collect}\ (\texttt{Collect}\ x) \dot{\rightarrow} \texttt{Collect}\ x)) \\
\Leftrightarrow\ & (\mathsf{Id}, \mathsf{Id}) \in \mathsf{Rel}_{[x \mapsto R]}\ (\texttt{Collect}\ (\texttt{Collect}\ x) \dot{\rightarrow} \texttt{Collect}\ x) \\
\Leftrightarrow\ & (\mathsf{Id}, \mathsf{Id}) \in (\mathsf{Rel}_{[x \mapsto R]}\ (\texttt{Collect}\ x) \leftarrow \mathsf{Rel}_{[x \mapsto R]}\ (\texttt{Collect}\ (\texttt{Collect}\ x))) \\
\Leftrightarrow\ & (\mathsf{Id}, \mathsf{Id}) \in (\mathsf{Rel}_{[x \mapsto R]}\ x \leftarrow \mathsf{Rel}_{[x \mapsto R]}\ x) \\
\Leftrightarrow\ & (\mathsf{Id}, \mathsf{Id}) \in (R \leftarrow R) \\
\Leftrightarrow\ & \forall a, b.\ (a, b) \in R \Rightarrow (\mathsf{Id}\ a, \mathsf{Id}\ b) \in R \\
\Leftrightarrow\ & \forall a, b.\ (a, b) \in R \Rightarrow (a, b) \in R \\
\Leftrightarrow\ & \mathsf{true}
\end{aligned}$$

□

Proposition 10 *If* $\sigma_{\mathrm{T}}\ \texttt{union} = \lambda a.\ \lambda b.\ a$ *and* $\kappa\ \texttt{Collect} = \mathsf{Id}$,

$$\mathbf{A3}: (\sigma_{\mathrm{T}}\ \texttt{union}, \sigma_{\mathrm{T}}\ \texttt{union}) \in \mathsf{Rel}\ ((\dot{\forall}(\lambda x.\texttt{Collect}\ x \dot{\rightarrow} \texttt{Collect}\ x \dot{\rightarrow} \texttt{Collect}\ x)$$

Proposition 11 *If* $\sigma_{\mathrm{T}}\ \texttt{map} = \lambda f.\lambda a.\ f\ a$ *and* $\kappa\ \texttt{Collect} = \mathsf{Id}$,

$$\mathbf{A4}: (\sigma_{\mathrm{T}}\ \texttt{map}, \sigma_{\mathrm{T}}\ \texttt{map}) \in \mathsf{Rel}\ (\dot{\forall}(\lambda x.\dot{\forall}(\lambda y.(x \dot{\rightarrow} y) \dot{\rightarrow} \texttt{Collect}\ x \dot{\rightarrow} \texttt{Collect}\ y)))$$

We can now develop a parametricity result for (SIM $\leftarrow$ SIM). If the transformer is defined using the abstract `Collect` interface to set-like structures, then it cannot access any of the functions that destroy monotonicity. For example, if the interpretation of `Collect` is finite sets, functions of type $\dot{\forall}(\lambda x.\ \texttt{TS Collect}\ x\ T_1\ T_2 \rightarrow \texttt{TS Collect}\ (x \dot{\times} x)\ T_3\ T_4)$ only have access to $\{\cdot\}$, $\cup$, $\bigcup$, and mappings—meaning that the functions of this type cannot use set-operators such as cardinality.

Proposition 12 *Given* $u :: \dot{\forall}(\lambda x.\ \texttt{TS Collect}\ (F\ x)\ T_2\ T_3)$,
$v :: \dot{\forall}(\lambda x.\ \texttt{TS Collect}\ (F\ x)\ T_2\ T_3)$ *and* $\kappa\ \texttt{Collect} = \mathsf{total}$*:*

$$(u, v) \in \mathsf{SIM}_R \text{ equals } (u, v) \in \mathsf{Rel}_{[x \mapsto R]}\ (\texttt{TS Collect}\ (F\ x)\ T_2\ T_3)$$

With this proposition, we can prove that any f with type $\dot{\forall}(\lambda x.\ \texttt{TS Collect}\ x\ T_1\ T_2 \rightarrow \texttt{TS Collect}\ (F\ x)\ T_3\ T_4)$ is monotonic with respect to SIM:

$$\begin{aligned}
(\llbracket f \rrbracket, \llbracket f \rrbracket) \in\ & \mathsf{Rel}\ (\dot{\forall}(\lambda x.\ \texttt{TS Collect}\ x\ T_1\ T_2 \dot{\rightarrow} \texttt{TS Collect}\ (F\ x)\ T_1\ T_2)) \\
\Leftrightarrow\ & \forall R.(\llbracket f \rrbracket, \llbracket f \rrbracket) \in \mathsf{Rel}_{[x \mapsto R]}\ (\texttt{TS Collect}\ x\ T_1\ T_2 \dot{\rightarrow} \texttt{TS Collect}\ (F\ x)\ T_3\ T_4) \\
\Leftrightarrow\ & \forall R, a, b.\ (a, b) \in \mathsf{Rel}_{[x \mapsto R]}\ (\texttt{TS Collect}\ x\ T_1\ T_2) \\
& \Rightarrow (\llbracket f \rrbracket\ a, \llbracket f \rrbracket\ b) \in \mathsf{Rel}_{[F \mapsto R]}\ (\texttt{TS Collect}\ (x \dot{\times} x)\ T_3\ T_4) \\
\Leftrightarrow\ & \forall R, a, b.\ (a, b) \in \mathsf{SIM}_R \Rightarrow (\llbracket f \rrbracket\ a, \llbracket f \rrbracket\ b) \in \mathsf{SIM}_{(R \times R)} \\
\Rightarrow\ & \forall a, b.\ (a, b) \in \mathsf{SIM} \Rightarrow (\llbracket f \rrbracket\ a, \llbracket f \rrbracket\ b) \in \mathsf{SIM} \\
\Leftrightarrow\ & (\llbracket f \rrbracket, \llbracket f \rrbracket) \in (\mathsf{SIM} \leftarrow \mathsf{SIM})
\end{aligned}$$

From this, we now know that transformers that use `Collect` and that are appropriately polymorphic in the state parameter are monotonic with respect to SIM.

5.5 Decomposing proofs with transformers expressed in λ^M

An advantage of writing models as the composition of transformers is that, when verifying one model against another, the proof can potentially be decomposed point-wise into proofs about transformer pairs.

Proposition 13 *If* $(f, f') \in (P \leftarrow Q)$ *and* $(g, g') \in (Q \leftarrow R)$ *then* $(f \circ g, f' \circ g') \in (P \leftarrow R)$

Proof.

$$
\begin{aligned}
& (g, g') \in (Q \leftarrow R) \wedge (f, f') \in (P \leftarrow Q) \\
\Leftrightarrow\ & [\forall a, b.(a, b) \in R \Rightarrow (g\ a, g'\ b) \in Q] \wedge [\forall c, d.(c, d) \in Q \Rightarrow (f\ c, f'\ d) \in P] \\
\Rightarrow\ & \forall a, b.(a, b) \in R \Rightarrow (f(g\ a), f'\ (g'\ b)) \in P \\
\Leftrightarrow\ & \forall a, b.(a, b) \in R \Rightarrow ((f \circ g)\ a), (f' \circ g')\ b) \in P \\
\Leftrightarrow\ & (f \circ g, f' \circ g') \in (R \leftarrow P)
\end{aligned}
$$

□

For example, if we are trying to prove that $(f \circ g, f' \circ g') \in (\mathsf{SIM} \leftarrow \mathsf{BISIM})$, one possible strategy is to prove $(f, f') \in (\mathsf{SIM} \leftarrow \mathsf{SIM})$ and $(g, g') \in (\mathsf{SIM} \leftarrow \mathsf{BISIM})$.

Proposition 14 *If* $(f, f') \in (R \leftarrow Q)$ *and* $(u, v) \in Q$ *then* $(f\ u, f'\ v) \in R$

Using this rule, we can prove $(f\ u, f'\ v) \in \mathsf{SIM}$ by proving $(f, f') \in (\mathsf{SIM} \leftarrow \mathsf{SIM})$ and $(u, v) \in \mathsf{SIM}$.

If f is appropriately typed and $f = f'$ then, with the Parametricity results that we have developed in this chapeter, we can potentially discharge $(f, f') \in (\mathsf{SIM} \leftarrow \mathsf{SIM})$ based on f's type.

5.6 Summary

In this chapter we have provided a methodology for decomposing and simplifying correctness proofs of transformer-based models. The methodology begins with breaking a proof down into smaller proofs—each of which can potentially require reasoning only about local extensions. Then, using the Parametricity-based simplification strategy, we have provided a way of discharging a limited class of resulting obligations.

The material in this chapter has demonstrated that transformers can potentially provide a new axis for decomposition and simplification during a proof of a microarchitectural design's correctness. If the forms of the specification and implementation align in the correct way, we can break down the proof obligation. The connection that we have demonstrated between simulation and Parametricity may also allow us to automatically discharge some of the proof obligations.

Chapter 6

Applying the theory of transformers

In Chapter 3 we surveyed several approaches that have been used when formally verifying a microarchitectural model against its ISA specification. In Chapter 5 we developed a decomposition and simplification strategy that can be used when verifying models against specifications written as the composition of transformers. In this chapter, we apply these techniques to the decomposition of a proof that, `wma`, the microarchitectural model from Chapter 4, is correct with respect to its ISA specification, `wa`.

Note that the focus of this chapter is on factoring the proof into obligations—not completely proving correctness. Therefore several of the proof obligations will not be proved rigorously. In these cases we will provide a reference to a technique from the literature which can solve the problem.

As we saw from the execution demonstration at the end of Chapter 4, the microarchitectural model can execute a program in a different number of cycles than ISA; meaning that it is not possible to prove that $(\texttt{wma}\ p, \texttt{wa}\ p) \in \mathsf{SIM}$. Therefore our aim is to work on the proof that $(\texttt{wma}\ p, \texttt{wa}\ p) \in \mathsf{FP}$. That is, we are trying to prove that for each flush-point trace in `wma` there exists an analogous trace in `wa` such that the observations of the flushed states are equivalent. Note that we are not proving $(\texttt{oma}\ p, \texttt{oa}\ p) \in \mathsf{FP}$—we will discuss this in the next chapter. Figure 6.1 displays the picture of the overall proof decomposition. Each step is numbered for discussion. The remainder of this chapter discusses these steps in more detail. A review and summary of the decomposition and proof steps will be given at the end of the chapter.

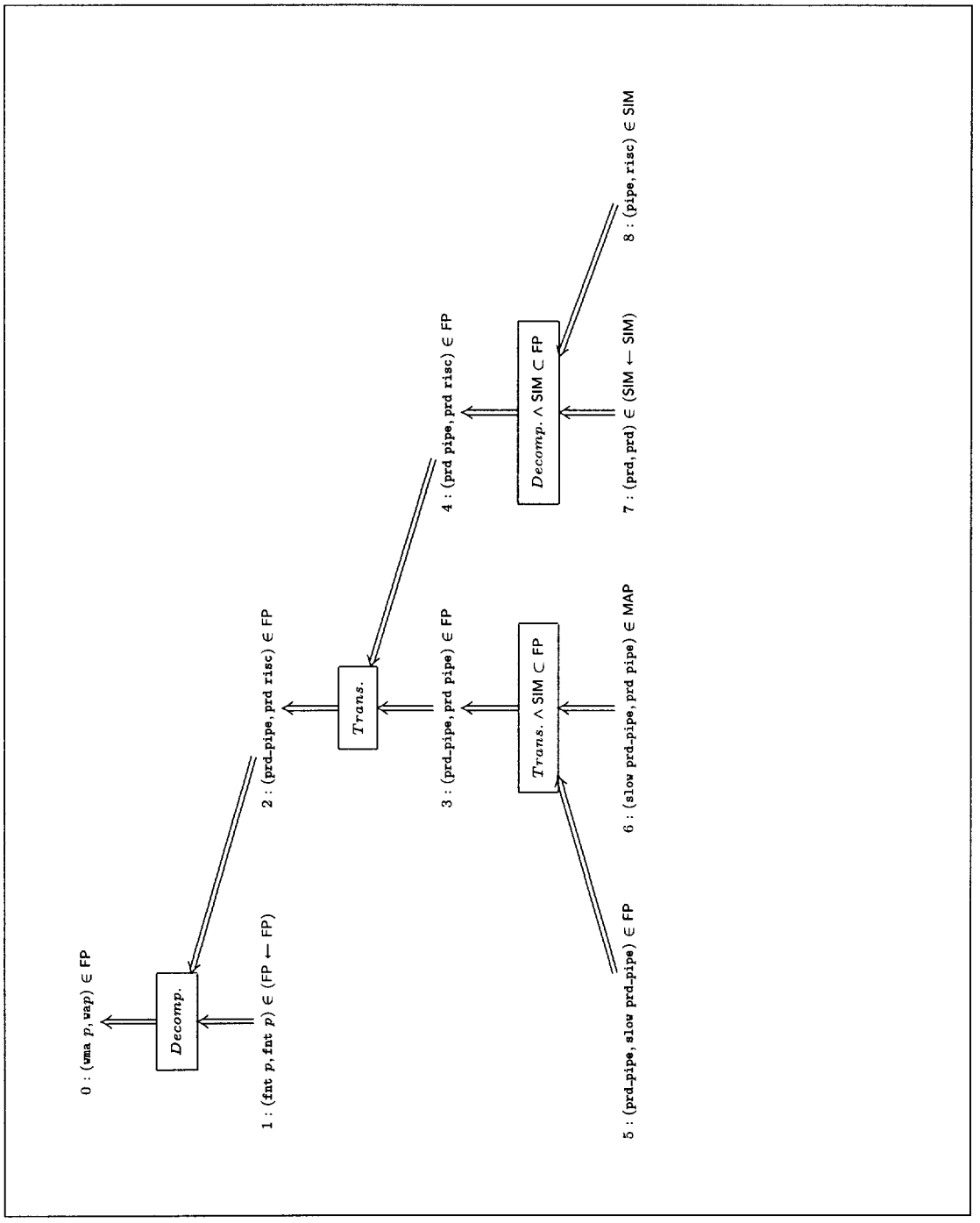


Figure 6.1: Top-level proof decomposition

6.1 Decomposing the proof into obligations

We begin the proof by decomposing $(\texttt{wma}\ p, \texttt{wa}\ p) \in \mathsf{FP}$ into several obligations. First, by unfolding the definitions of `wma` and `wa` we have:

$$(\texttt{fnt}\ p\ \texttt{prd\_pipe}, \texttt{fnt}\ p\ (\texttt{prd}\ \texttt{risc})) \in \mathsf{FP}$$

Using Proposition 3 from Chapter 5 we can decompose this into two obligations:

$$1 : (\texttt{fnt}\ p, \texttt{fnt}\ p) \in (\mathsf{FP} \leftarrow \mathsf{FP})$$

$$2 : (\texttt{prd\_pipe}, \texttt{prd}\ \texttt{risc}) \in \mathsf{FP}$$

6.2 Proving Obligation 1: $(\texttt{fnt}\ p, \texttt{fnt}\ p) \in (\mathsf{FP} \leftarrow \mathsf{FP})$

This obligation essentially states that if there exists an R such that $(u, v) \in \mathsf{FP}_R$ then $(\texttt{fnt}\ p\ u, \texttt{fnt}\ p\ v) \in \mathsf{FP}$. Let `Bool` be an instance of the `Bubble` type-class where `True` is the `bubble` instruction. Let the same R be the witness for $(\texttt{fnt}\ p\ u, \texttt{fnt}\ p\ v) \in \mathsf{FP}$. By the definition of `fnt` we know that conditions $\mathsf{FP}.A$ and $\mathsf{FP}.C$ from Definition 6 hold. We must now prove $\mathsf{FP}.B$. Imagine that the following is a flush point trace of $\texttt{fnt}\ p\ u$:

$$s_0 \xrightarrow[\texttt{fnt}\ p\ u]{b_0} s_1 \xrightarrow[\texttt{fnt}\ p\ u]{b_1} \cdots \xrightarrow[\texttt{fnt}\ p\ u]{b_m} s_m$$

where $b_* :: \texttt{Bool}$. This induces a flush-point trace in the underlying system u:

$$s_0 \xrightarrow[u]{i_0} s_1 \xrightarrow[u]{i_1} \cdots \xrightarrow[u]{i_n} s_m$$

where $i_0 \ldots i_n$ are chosen by `fnt` from p. By $(u, v) \in \mathsf{FP}$, we know that there exists a flush-point trace in v:

$$s'_0 \xrightarrow[v]{i'_0} s'_1 \xrightarrow[v]{i'_1} \cdots \xrightarrow[v]{i'_n} s'_n$$

such that $i' \in \Gamma\ i$, $\texttt{observe}\ u\ s_0 = \texttt{observe}\ v\ s'_0$ and $\texttt{observe}\ u\ s_m = \texttt{observe}\ v\ s'_n$. We also know that there exists a q such that $\forall 0 \leq k < n - q.\ \texttt{stalled}(s_k) \Leftrightarrow i'_k = \texttt{bubble}$ and $\forall n - q \leq k < n.\ i'_k = \texttt{bubble}$. Let b' be the sequence defined by the function $\lambda k.\ k < q$. By the definition of `fnt` we know that feeding b' to the transition system $\texttt{fnt}\ p\ v$ will cause

it to feed the sequence s' to v in the underlying system. Since s' is a flush-point trace of v, we know that s' is a flush-point trace of $\texttt{fnt}\ p\ v$ and that

$$\begin{aligned}
&\forall 0 \leq k < n - q.\ \texttt{stalled}(s'_k) \Leftrightarrow b'_k = \texttt{bubble} \\
&\wedge \forall n - q \leq k < n.\ b'_k = \texttt{bubble} \\
&\wedge\ \forall 0 \leq k < n.\ s'_{k+1} \in [\![\texttt{next}]\!]\ v\ i'_k\ s'_k
\end{aligned}$$

Therefore, $(\texttt{fnt}\ p\ u, \texttt{fnt}\ p\ v) \in \mathsf{FP}$.

6.3 Proving Obligation 2: $(\texttt{prd\_pipe}, \texttt{prd risc}) \in \mathsf{FP}$

In this part of the proof we are essentially showing that the execution core of the microarchitectural model is correct with respect to the execution core of the architectural model. This is difficult to prove directly because, as we discussed in Chapter 4, `prd risc` implements predication at the front-end of `risc`, while `prd_pipe` places the predication check at the end of the pipeline. One approach to solving this problem is to break the obligation down into two simpler ones: one that proves that the predication code in `prd_pipe` is correct with respect to `prd`; and the other that proves that the non-predicated parts of `prd_pipe` are correct with respect to `risc`.

We can do this by introducing an intermediate model. Because the specification is in transformer form, we use a part of the specification to build an intermediate model: `prd pipe`. With the transitivity of FP we know that, $(\texttt{prd\_pipe}, \texttt{prd risc}) \in \mathsf{FP}$ is implied by Obligations 3 and 4:

$$3 : (\texttt{prd\_pipe}, \texttt{prd pipe}) \in \mathsf{FP}$$

$$4 : (\texttt{prd pipe}, \texttt{prd risc}) \in \mathsf{FP}$$

6.4 Proving Obligation 3: $(\texttt{prd\_pipe}, \texttt{prd pipe}) \in \mathsf{FP}$

In essence, this obligation states that pipelining with predication implemented in the back-end of a pipeline is correct with respect to pipelining with predication implemented in the front-end. As discussed in Chapter 4, the distinction between the two techniques is that the transition system with predication in the back-end does not stall on an `R2P`

instruction. If it weren't for this difference we could easily build a simulation relation between them. However, if we were to force `prd_pipe` to stall for three cycles whenever it received an `R2P` instruction, it would simulate `prd pipe`. Figure 6.2 contains the definition of a transformer, called `slow`, which does exactly this. Essentially `slow` takes a model with state space `s` and constructs a new model that consists of `s` paired with two prophecy variables: `(s, Int, Int)`. The second element keeps track of how many cycles to stall, and the third records how many cycles the machine has been stalling. On every cycle in which the prophecy variables are set to 0, `slow` uses the function `slowp` to determine how many cycles to stall based on the incoming instruction and the state of the underlying machine. This is similar to tricks used by Jones [41, 42], Jones *et al.* [44] and Abadi & Lamport [4]. Lines 16 through 19 of Figure 6.2 check for the case that either a `bubble` instruction or an instruction that cannot exist in the `prd pipe` pipeline is found in the slowed down predicated pipeline. In this case, the number of cycles to stall is adjusted such that the execution of the two pipelines match. We can construct an intermediate model by applying `slow` to `prd_pipe`. Then, using transitivity of FP and SIM $\subseteq$ FP we can decompose Obligation 3 into:

$$5 : (\texttt{prd\_pipe}, \texttt{slow prd\_pipe}) \in \mathsf{FP}$$

$$6 : (\texttt{slow prd\_pipe}, \texttt{prd pipe}) \in \mathsf{SIM}$$

6.5 Proving Obligation 4: (prd pipe, prd risc) $\in$ FP

By applying the decomposition rule in the same fashion as we decomposed the top-level obligation, (`prd pipe, prd risc`) $\in$ FP can be broken down into:

$$7 : (\texttt{prd}, \texttt{prd}) \in (\mathsf{SIM} \leftarrow \mathsf{SIM})$$

$$8 : (\texttt{pipe}, \texttt{risc}) \in \mathsf{SIM}$$

```
0  slow = slowdown slowp
1
2  slowdown :: (Collect c,Bubble i) => (i-> (s,Int,Int) -> Int) ->
3              TS c i s (Obs o) -> SM c i (s,Int,Int) (Obs o)
4  slowdown p m = (int,nxt,ob)
5           where
6           int = do {s <- initial m; return (s,0,0)}
7           nxt i (s,0,k) = do {s' <- next m i s
8                               ; return (s',p i (s,0,k),p i (s,0,k))
9                               }
10          nxt i (s,n,k) = do {s' <- next m bubble s; return (s',n-1,k)}
11          ob (s,0,_) = observe m s
12          ob (s,n,_) = r_stalled
13
14
15 slowp (R2P _ _ _) ( ((rf , (p, q, r)), (prf, other)) ,n,n')
16          | n==0 && n' > 0 = 0
17          | bub p prf3 && bub q prf2 && bub r prf1 = 0
18          | bub p prf3 && bub q prf2 = 1
19          | bub p prf3 = 2
20          | otherwise = 3
21              where
22              bub p env = not (pred_true_in p env)
23                        || p==prd_bubble_trans || is_r2p p
24                        || is_set p
25              ss = ((rf,(p,q,r)), (prf, other))
26              ((_, _),(prf3, _)) = run_ prd_pipe b3 ss
27              ((_, _),(prf2, _)) = run_ prd_pipe b2 ss
28              ((_, _),(prf1, _)) = run_ prd_pipe b1 ss
29          where s = ( ((rf , (p, q, r)), (prf, other)) ,n,n')
30                b1 = [bubble]
31                b2 = [bubble,bubble]
32                b3 = [bubble,bubble,bubble]
33 slowp i _             = 0
```

Figure 6.2: slow: A prophecy-variable based transformer

6.6 Proving Obligation 5: (prd_pipe, slow prd_pipe) ∈ FP

Assuming that a given transition system m is deterministic, let the function V be defined accordingly[1]:

$$\mathtt{V}_m\ s = \mathtt{next}_m\ \mathtt{bubble}\ (\mathtt{next}_m\ \mathtt{bubble}\ (\mathtt{next}_m\ \mathtt{bubble}\ s))$$

Assume that we can prove the following property of prd_pipe:

$$\forall s, s'.\ \mathtt{V}_{\mathtt{prd\_pipe}}\ s = \mathtt{V}_{\mathtt{prd\_pipe}}\ s' \Rightarrow \mathtt{V}_{\mathtt{prd\_pipe}}\ (\mathtt{next}_{\mathtt{prd\_pipe}}\ \mathtt{bubble}\ s) = \mathtt{V}_{\mathtt{prd\_pipe}}\ s'$$

Call this property *Bubble independence.* Also assume that we can also prove that the Burch-Dill verification condition holds for prd_pipe when compared to itself.

$$\forall s, s', i.\ \mathtt{V}_{\mathtt{prd\_pipe}}\ s = \mathtt{V}_{\mathtt{prd\_pipe}}\ s' \Rightarrow \mathtt{V}_{\mathtt{prd\_pipe}}\ (\mathtt{next}\ i\ s) = \mathtt{V}_{\mathtt{prd\_pipe}}\ (\mathtt{next}\ i\ s')$$

Call this property *Self Burch-Dill.*

These two properties are like those that are often proved in the literature using a validity checker such as CVC-lite [11] or Zapato [10]. Examples include [12, 44, 42, 41, 43, 60, 45, 46]

Let R be defined as:

$$\mathtt{R}\ (s, (s', n, k)) \triangleq n = 0 \wedge \mathtt{V}_{\mathtt{prd\_pipe}}\ s = \mathtt{V}_{(\mathtt{slow\ prd\_pipe}}\ s'$$

If the two properties above hold then we can now prove (prd_pipe, slow prd_pipe) ∈ $\mathsf{FP}_{\mathtt{R}}$. Let i be a sequence of instructions such that the following is a flush-point trace:

$$s_0 \xrightarrow[\mathtt{prd\_pipe}]{i_0} s_1 \xrightarrow[\mathtt{prd\_pipe}]{i_1} \cdots \xrightarrow[\mathtt{prd\_pipe}]{i_n} s_n$$

Consider each of these transtions: $s_k \xrightarrow[\mathtt{prd\_pipe}]{i_k} s_{k+1}$. Using mixtures of the self Burch-Dill property and Bubble independance we can prove that, if $\mathtt{V}_{\mathtt{prd\_pipe}}\ s_k = \mathtt{V}_{\mathtt{prd\_pipe}}\ s'_k$ then the following four conditions hold[2]:

- $\mathtt{V}\ (\mathtt{next}\ i_k\ s_{k+1}) = \mathtt{V}\ (\mathtt{next}\ i_k\ s'_{k+1})$

[1] This could be generalized to non-deterministic systems, but that is unnecissary in this case

[2] We are assuming that prd_pipe is the underlying machine in these conditions

- $\mathtt{V}\ (\mathtt{next}\ i_k\ s_{k+1}) = \mathtt{V}\ (\mathtt{next\ bubble}\ (\mathtt{next}\ i_k\ s'_{k+1}))$
- $\mathtt{V}\ (\mathtt{next}\ i_k\ s_{k+1}) = \mathtt{V}\ (\mathtt{next\ bubble}\ (\mathtt{next\ bubble}\ (\mathtt{next}\ i_k\ s'_{k+1})))$
- $\mathtt{V}\ (\mathtt{next}\ i_k\ s_{k+1}) = \mathtt{V}\ (\mathtt{next\ bubble}(\mathtt{next\ bubble}\ (\mathtt{next\ bubble}\ (\mathtt{next}\ i_k\ s'_{k+1}))))$

By abstracting `slowp` we can safely assume that the result of `slowp` applied to i_k is $m \in \{0,1,2,3\}$. We know that this will induce a sequence in the machine `slow prd_pipe` such that

$$((s'_k, 0, \_) \xrightarrow[\mathtt{slow\ prd\_pipe}]{i_k} (s'_{k+1}, m, m) \xrightarrow[\mathtt{slow\ prd\_pipe}]{\mathtt{bubble}} \cdots \xrightarrow[\mathtt{slow\ prd\_pipe}]{\mathtt{bubble}} (s'_{k+m}, 0, m)$$

By the four conditions above and the definition of `slow` we can therefore conclude that, if $\mathtt{V}\ s_k = \mathtt{V}\ s'_k$, then

- $\mathtt{V}\ s_{k+1} = \mathtt{V}\ s'_{k+m} = \mathtt{V}_{(\mathtt{slow\ prd\_pipe})}\ (s_{k+m}, 0, m)$
- $\mathtt{stalled}_{\mathtt{prd\_pipe}}\ s_{k+1} \Leftrightarrow \mathtt{stalled}_{(\mathtt{slow\ prd\_pipe})}\ s'_{k+m}$
- $\mathtt{stalled}_{\mathtt{prd\_pipe}}\ s_{k+1} \Leftrightarrow \mathtt{stalled}_{(\mathtt{slow\ prd\_pipe})}\ (s'_{k+m}, 0, m)$
- $\forall 0 < j \leq m.\ \mathtt{stalled}_{(\mathtt{slow\ prd\_pipe})}\ (s'_{k+j}, j, m)$

Let f be the function that maps an instruction j to j followed by value sequence of bubbles. Assume that the number of bubbles is equal to the value returned `slowp` when applied to j. Let i' be the instruction sequence constructed from i by a concatenation of the sequences $f(i)$. This leads to a flush-point trace in `slow prd_pipe`, proving that $(\mathtt{prd\_pipe}, \mathtt{slow\ prd\_pipe}) \in \mathsf{FP}$:

$$((s'_0, 0, \_) \xrightarrow[\mathtt{slow\ prd\_pipe}]{i'_0} \cdots (s'_l, 0, m)$$

6.7 Proving Obligation 6: $(\mathtt{slow\ prd\_pipe}, \mathtt{prd\ pipe}) \in \mathsf{SIM}$

An important distinction between `slow prd_pipe` and `prd pipe` is that the pipeline registers in `slow prd_pipe` hold `PTrans` values, whereas the pipeline registers in `prd pipe` hold `Trans` values. Therefore, when a predicated instruction i occurs in a `slow prd_pipe`

pipeline register, the analogous register in `prd pipe` is a function of what `prd` did with i when it was placed into the machine.

The Obligation $(\texttt{slow prd\_pipe}, \texttt{prd pipe}) \in \mathsf{SIM}$ can be proved with the simulation mapping in Figure 6.3. This function computes the *underlying transaction* from a `PTrans` value.

The mapping uses the prophecy variable in `slow prd_pipe` to calculate the analogous state in `prd pipe`. In the case that the prophecy variable is 3 (lines 0 through 3), on the last cycle the `R2P` instruction was issued and therefore:

- The `nw` register in `prd pipe` must be an empty transaction.
- The `rd` register in `prd pipe` is the underlying transaction in `rd` from `slow prd_pipe`. However, because the transaction in `wb` can potentially be an `R2P` or `SET`, we must calculate the underlying transaction with the predicate register file updated by any `R2P` or `SET` instructions in `wb`.
- The `wb` register in `prd pipe` is the underlying transaction from `wb` in `slow prd_pipe`.
- Because the last instruction issued to `slow prd_pipe` was an `R2P` the latch is set to the underlying instruction from `nw`.

Similar logic is used in lines 4 through 13. Lines 4 through 7 cover the case that the prophecy variable is 2. Lines 8 through 13 cover the case that the prophecy variable is 1.

In the case that the prophecy variable is 0, the mapping searches for `R2P` instructions in the pipeline registers of `slow prd_pipe`. Lines 15 through 19 cover the case where there is an `R2P` instruction in `nw`. In this case the machine received an `R2P` instruction four cycles before this one. Therefore the analogous state in `prd pipe` must be completely flushed.

In the other cases (lines 20 through 38) there is no `R2P` instruction in `nw`. If there is an `R2P` instruction in `rd` (lines 20 through 25) then the machine was issued an `R2P` instruction five cycles before this one, and the analogous state in `prd pipe` must be only be partially flushed.

Notice how, in many of the cases, `SET` instructions are searched for with `updt'` when calculating the predicate register file. This is because, in `prd pipe`, the effect of `SET`

instructions are seen as soon as they are issued.

Intuitively speaking, the function `mapping` accelerates the execution `slow prd_pipe` in cases where it is necissary in order that the observations of the two machines are equivilent. We can apply symbolic simulation to establish the conditions required to prove SIM.

To prove that `mapping` is a simulation relation we must prove that

SIM.A) $\forall a, b.\ (a, b) \in$ `mapping` $\Rightarrow$ `observe` $u\ a =$ `observe` $v\ b$

To prove this we could use a validity checker—as we also proposed when discussing Obligation 6. The proof of this condition could even be decomposed into seven validity checks: one for each of the cases that comprises the function `mapping`.

SIM.B) $\forall a \in$ `initial` $u.\ \exists b \in$ `initial` $v.\ (a, b) \in$ `mapping`

To prove this we can simply symbolically execute `mapping` on the single initial state of `slow prd_pipe`. This state is

```
(((initial_rf,flushed_pipe),(inital_prf,()),0,0))
```

where

```
flushed_pipe = (prd_bubble_trans,prd_bubble_trans,prd_bubble_trans)
```

After executing `mapping` (`slow prd_pipe`) we the single initial state of `prd pipe`:

```
((initial_rf,flushed_pipe'),(initial_prf,Nothing))
```

where

```
flushed_pipe' = (bubble_trans,bubble_trans,bubble_trans)
```

SIM.C) $\forall a, a', b, i.\ [(a, b) \in$ `mapping` $\wedge\ a' \in$ `next` $u\ i\ a] \Rightarrow [\exists b'.\ b' \in$ `next` $v\ i\ b \wedge (a', b') \in$ `mapping`]

To prove this condition we could again rely on a validity checker.

```
0  mapping (((rf,(nw, rd, wb)),(prf, ())), 3,n')
1    = ((rf ,(bub_trans,underl' prf1 rd,underl' prf wb))
2      ,(updt [rd,wb] rf prf ,Just (instr nw)))
3          where prf1 = updt [wb] rf prf
4  mapping (((rf,(nw, rd, wb)),(prf, ())), 2,n')
5    = ((rf ,(bub_trans,bub_trans,underl' prf wb))
6      ,( updt' [nw,rd] (updt [wb] rf prf)
7       ,Just (if is_r2p nw then instr nw else instr rd)))
8  mapping (((rf,(nw, rd, wb)),(prf, ())), 1,n')
9    = ((rf ,(bub_trans,bub_trans,bub_trans))
10     ,(updt' [nw,rd,wb] prf
11      ,Just (if is_r2p nw then instr nw
12             else if is_r2p rd then instr rd
13             else instr wb)))
14 mapping (((rf,(nw, rd, wb)),(prf, ())), 0,n')
15   | is_r2p nw = let (((rf',_),(prf3, _)), _,_) = run_ prd_pipe' b3 s
16                     b3 = [bubble,bubble,bubble]
17                in   ((rf' ,(bub_trans,bub_trans,bub_trans))
18                      ,(prf3 ,Nothing)
19                      )
20   | is_r2p rd = let (((rf',_),(prf2, _)), _,_) = run_ prd_pipe' b2 s
21                     b2 = [bubble,bubble]
22                     prf3 = updt' [nw] prf2
23                in   ((rf' ,(underl' prf2 nw,bub_trans,bub_trans))
24                      ,(prf3,Nothing)
25                      )
26   | is_r2p wb = let (((rf',_),(prf1, _)), _,_) = run_ prd_pipe' b1 s
27                     b1 = [bubble]
28                     prf3 = updt' [nw,rd] prf1
29                     prf2 = updt' [rd] prf1
30                in  ((rf',(underl' prf2 nw,underl' prf1 rd,bub_trans))
31                      ,(prf3,Nothing)
32                      )
33   | otherwise = let prf3 = updt' [nw,rd,wb] prf
34                      prf2 = updt' [rd,wb] prf
35                      prf1 = updt' [wb] prf
36                 in ((rf ,(underl' prf2 nw,underl' prf1 rd,
37                       underl' prf wb)),(prf3,Nothing)
38                    )
39     where s = (((rf,(nw, rd, wb)),(prf, ())), 0,n')
```

Figure 6.3: mapping: witness to (slow prd_pipe, prd pipe) $\in$ MAP

```
0  instr (_,_,i,_,_) = i
1
2  underl prf ( dsts, _ , (GO j), srcs, _)=(dsts, j, srcs)
3  underl prf ( dsts, _ , (IF j _), srcs, _)=(dsts, j, srcs)
4  underl prf ((x, _):d,[],P2R _ _,_,[(Just y,_)])
5    = mkTrans (CNT x (if (readEnv prf y) then 1 else 0))
6  underl prf _ = bubble_trans
7
8  underl' prf i = if pred_true_in i prf then underl prf i
9                      else bubble_trans
10
11 updt [] rf prf = prf
12 updt (x:xs) rf prf
13  = case instr x of
14         R2P r1 r2 r' -> let e1 = updateEnv e (r1,readEnv rf r' /= 0)
15                             e2 = updateEnv e1 (r2,readEnv rf r' == 0)
16                         in e2
17         SET r v -> updateEnv e (r,v)
18         otherwise -> e
19         where e = updt xs rf prf
20
21 updt' [] prf = prf
22 updt' (x:xs) prf
23  = case instr x of
24         SET r v -> updateEnv e (r,v)
25         otherwise -> e
26         where e = updt' xs prf
```

Figure 6.4: Functions used in the definition of mapping

6.8 Proving Obligation 7: $(\texttt{prd}, \texttt{prd}) \in (\mathsf{SIM} \leftarrow \mathsf{SIM})$

This obligation states that prd is monotonic with respect to (SIM ← SIM). Recall that in Chapter 4, prd's type was given as:

```
(Collect c, Bubble i, Eq r, Eq w, Bind i r w, Integral w) =>
    TS c i s (Obs (Env r w)) ->
    TS c (Prd_Instr i r r) (Prd_St s r (Prd_Instr i r r)) (Obs (Env r w))
```

Because the TS type corresponds via the Parametricity theorem to the SIM relation (as described in Chapter 5), we can use this theorem to establish Obligation 7.

Assuming that T_1, T_2, and T_3 meet the constraints of the type classes, we can encode prd in λ^M as a function with type:

$$\begin{aligned}\dot{\forall}(\lambda x.\quad & \texttt{TS Collect } T_1\ x\ (\texttt{Obs (Env } T_2\ T_3))\dot{\rightarrow} \\ & \texttt{TS Collect (Prd\_Instr } T_1\ T_2\ T_2) \\ & \qquad (\texttt{Prd\_St } x\ T_2\ (\texttt{Prd\_Instr } T_1\ T_2\ T_2))\ (\texttt{Obs (Env } T_2\ T_3)))\end{aligned}$$

Because prd is polymorphic in its state-type and can be expressed in λ^M, by Parametricity we know that $(\texttt{prd}, \texttt{prd}) \in (\mathsf{SIM} \leftarrow \mathsf{SIM})$.

6.9 Proving Obligation 8: $(\texttt{pipe}, \texttt{risc}) \in \mathsf{SIM}$

This problem has been well addressed in the formal verification literature, as discussed in Section 3.6. To prove $(\texttt{pipe}, \texttt{risc}) \in \mathsf{SIM}$ we could use any of these published techniques.

As an example, we could use Burch & Dill's technique [17] of automatically constructing the simulation relation with the next functions of the two machines, together with the bubble instruction. As Burch & Dill do, we could then use a validity checker to prove condition SIM.C. We could also use the validity checker to prove conditions SIM.A and SIM.B.

6.10 Summary

In this chapter we have decomposed and discharged the top level proof obligation $(\texttt{wma}\ p, \texttt{wa}\ p) \in \mathsf{FP}$. This decomposition was performed using the following strategies:

Transformer decomposition: This strategy was applicable in three points of the overall proof decomposition. What's more: the decomposition rules left proof obligations in several cases that were naturally solved by other techniques.

Parametricity: We were able to apply the Parametricity theorem to Obligation 7. This, combined with an application of Proposition 4 and the well-known techniques for proving RISC pipelines correct, discharges a significant part of the correctness proof.

Transitivity with intermediate models: The strategy of building intermediate models which more closely model the specification was applicable in the decomposition of Obligations 2 and 3.

Strengthening with the correctness criteria hierarchy: In several cases we used the connections between the correctness criteria developed in Chapter 3 to simplify the proofs. Namely, we used the result that $\mathsf{SIM} \subset \mathsf{FP}$ when the domain of these relations is restricted to initially flushed systems.

Transformer modeling style: We were able to mix and match pieces of the microarchitectural design and the ISA to build an intermediate model during the decomposition of Obligation 2. The clean interfaces encouraged by the transformer modeling method helped facilitate this.

Chapter 7

Conclusion

As microprocessors have grown more complex, researchers have addressed the problem of verifying the underlying designs with a number of new proof techniques, many of which make use of structure inherent in the superscalar and out-of-order execution cores. Meanwhile, microprocessor designers have not only been improving the execution cores of microprocessors, they have also been adding optimizations to the front-ends. These front-end features are typically in the form of instruction-set extensions, examples of which include parallelism annotations and predication.

One purpose of this dissertation has been to provide a modular method of modeling these instruction-set extensions. When modeled in this way, we have set out to demonstrate that the extra structure can be exploited in the decomposition of a microarchitectural correctness proof. In this dissertation we have introduced a method of modeling instruction-set extensions. We have argued that instruction-set extensions, when modeled in this style, can provide a new axis for proof decomposition. We have also demonstrated that this axis of decomposition can work hand-in-hand with other known proof techniques, such as uninterpreted functions, intermediate models, and Burch & Dill's flush-based simulation mapping.

7.1 Conclusions

In this section we draw several conclusions from the work in this dissertation.

Transformers facilitate modularity in design

When adding an extension to an instruction-set architecture, many decisions must be made about the interactions between features. For example, when adding predication and explicit parallelism, we could have chosen to implement one of the following possible interactions:

A) Instructions within basic blocks are predicated, but not basic blocks themselves.

B) Instructions are not predicated, but basic blocks are.

C) Both instructions and basic blocks are predicated.

Using the modularity that is inherent in transformer-styled modeling, we are encouraged to make these distinctions clear and precise. Predication was formally defined in Chapter 4 in isolation of explicit parallelism. That is: `prd` is the definition of predication. We can then combine this feature with explicit parallelism in a number of ways. For example, in Chapter 4 we chose to implement **A** from above with:

```
cnc 1 (prd risc)
```

Had we wanted to implement choice **B**, we could have used:

```
prd (cnc 1 risc)
```

Choice **C** would be:

```
prd (cnc 1 (prd risc))
```

In a more monolithic modeling style, the artifacts of these decisions would be embedded in the code and difficult to manipulate. This demonstrates how transformer-based modeling encourages the development of modular designs. To the best of our knowledge, this sort of modeling method has not been used before for instruction-set specification—and the power we get from the method has not been seen until now.

Transformer based proof techniques complement more traditional proof techniques

When new modeling and proof methods are introduced in the literature, they are often intended as replacements for old techniques. The work presented in this dissertation, in contrast, adds a new axis for proof decomposition. All of the previously known techniques can still be applied. That is, the transformer decomposition rule and Parametricity can work in unison with other verification techniques such as uninterrupted functions, intermediate models, and abstraction.

As an example, consider the case in Chapter 6 where we used the transformer decomposition rule to decompose the obligation $(\texttt{prd pipe}, \texttt{prd risc}) \in \mathsf{FP}$. The resulting obligations could then be discharged using a combination of Burch & Dill's mapping, prophecy variables, and Parametricity. This case demonstrates how transformer-based modeling and decomposition complement the set of common microarchitectural proof techniques.

Transformers facilitate the construction of intermediate models

We have seen that the transformer-based modeling style presents certain challenges. For example: the orthogonality that transformers impose made modeling a fast predicated pipeline with transformers difficult in Chapter 4. The cause of the difficulty was that the optimization that we implemented required structural information that crossed the boundaries of the transformer `prd` and `pipe`. The discipline of transformer-based modeling did, however, provide us with a benefit when trying to decompose the proof of correctness: we were able to combine parts from different worlds to construct the intermediate model `prd pipe`.

Parametricity provides free monotonicity results for some correctness criteria

Using Parametricity, we discharged Obligation 7 based on the type of `prd`. Because this result was not dependent on the definition of `prd`, the proof of its monotonicity holds even when the definition is modified—so long as the type remains appropriately polymorphic.

Although it was not required in this dissertation, a similar theorem holds for `fnt`. Based on the type of `fnt`, and not its definition, we know that it is monotonic with respect

to both simulation and bisimulation. This result could be used to structure and decompose a number of proofs described in the literature that use microarchitectural models with built-in front-ends. Had we used transformers to specify speculative loads or multimedia instructions, we could probably also have demonstrated a type-based monotonicity result for these transformers.

7.2 Future work

In this section we discuss several issues not addressed elsewhere in the dissertation and outline some directions for future research.

7.2.1 Machine checking the proof decomposition

The focus of this dissertation has been on the reasoning and organization of microarchitectural proofs—not on the software tools used during a proof. But the question naturally arises: How amenable is this work to a machine checked or machine guided proof system?

The answer is that, if the intention is to apply this work to traditional monolithic specifications and microarchitectural implementations, it would probably be difficult to automate the process in software with tactics or specialized routines. This is because the approach requires that the model and specification are both presented in an unusual and stylized form. The difficult aspect of this method is formulating the model and specification in the necessary form, not the reasoning behind a proof decomposition.

In this work we have used both higher-order functions and polymorphism. If we were trying to use a machine-based verification system to check reasoning like the work presented in this dissertation, we would need to use a system, such as HOL or PVS, that supports higher-order logic and polymorphism.

7.2.2 Algorithmically proving the obligations

An important issue that is not addressed in this dissertation is the algorithmic discharging of the proof obligations. At several points in Chapter 6 we simply left references to applicable techniques and tools, and presented several proof sketches. Future development

of this work should further explore the connection to these suggested tools and techniques and the remaining proof obligations.

7.2.3 Stream-based models

Streams [19, 49, 50] are an alternative formalism to transition systems. A stream is a function from time to a value. For example, our RISC instruction-set architecture could be modeled as a function with the type:

```
Int -> Opcode -> Obs RF
```

In this dissertation we have chosen to ignore streams for two reasons: First, the transition systems formulation is standard in the processor verification literature. Secondly, neither formalism is more or less powerful. In fact, previous work [21] has demonstrated that verification results in the streams formulation can be imported into the transition systems formulation, and vice versa.

In the literature on stream-based modeling, it is common to use higher-order functions; whereas in the transition system setting it is unusual. That said, one possible future direction for continued research would be to consider writing higher-order functions analogous to the transformers `prd` and `cnc` in a stream-based formalism.

7.2.4 Demonstrating that decomposition is helpful

The thesis of this dissertation assumes that decomposition is always good. The truth, in fact, is considerably more subtle. In practice, decomposition is certainly good when no other technique will work. But other techniques, such as abstraction, are typically best applied before resorting to decomposition. Decomposition still plays an important role in formal verification and specifically in formal microarchitectural verification. In particular, decomposition can help significantly when paired with powerful discharge rules for the decomposed obligations—which is the case in this dissertation. One avenue for further research would be to conduct a quantitative study of the effect of this work. For example, in the case of `prd_pipe`, is the verification route via Proposition 3 and decomposition better than using an algorithmic tool such as a model checker?

7.2.5 Alternative correctness criteria

In this dissertation we have essentially only considered connections between the correctness criteria used in the verification work: SIM and FP. However, in practice, many other criteria are available. These include well-founded bisimulation, trace containment, and a plethora of unnamed criteria that look similar to simulation. A natural question to ask is: how do these criteria relate? And, can we use Parametricity to prove monotonicity results for any of these criteria?

7.2.6 Liveness

In this dissertation we have only considered techniques for proving a class of correctness criteria called *safety properties*. That is, our research is applicable to criteria that demonstrates that nothing bad ever happens. We have not concerned ourselves with criteria specifying that progress is always made. It is possible, however, to encode any liveness property of a model using an intermediate model with prophecy variables and a safety property. Further research could explore the connection between the ideas presented in this dissertation and the modeling of liveness with prophecy variables.

7.2.7 Architectural relevance

Another question that should be explored with continued research is: beyond concurrent and predicated instructions, what can transformers specify? Further research could explore the boundaries of transformers—developing transformers that specify other architectural phenomena such as multimedia extensions, operating system support instructions, speculative loads, or rotating register-files. Another issue that could be addressed is that of proving $(\texttt{oma}\ p, \texttt{oa}\ p) \in \mathsf{FP}$. The difficulty here is in finding ways to prove $(\texttt{cnc}\ 3\ p, \texttt{cnc}\ 1\ p) \in (\mathsf{FP} \leftarrow \mathsf{FP})$.

Further research could also explore whether or not transformers, and the theory that facilitates their decomposition, can model other interesting microarchitectural optimizations—such as branch predication, multithreading or trace caches.

Bibliography

[1] AAGAARD, M., COOK, B., DAY, N., AND JONES, R. B. A framework for microprocessor correctness statements. *The International Journal on Software Tools for Technology Transfer* (2002), 298–312.

[2] AAGAARD, M., AND LEESER, M. Reasoning about pipelines with structural hazards. In *Conference on Theorem Provers in Circuit Design* (Bad Herrenalb, Germany, Aug. 1994), pp. 13–32.

[3] AAGAARD, M. D., COOK, B., DAY, N., AND JONES, R. B. A framework for microprocessor correctness statements. In *Conference on Correct Hardware Design and Verification Methods* (Edinburgh, United Kingdom, Sept. 2001), pp. 433–448.

[4] ABADI, M., AND LAMPORT, L. The existence of refinement mappings. *Theoretical Computer Science 2*, 82 (1991), 253–284.

[5] ALLEN, J., KENNEDY, K., PORTERFIELD, C., AND WARREN, J. Conversion of control dependence to data dependence. In *Symposium on Principles of Programming Languages* (Austin, Texas, Jan. 1983), pp. 177–189.

[6] ANDERSSON, G., BJESSE, P., COOK, B., AND HANNA, Z. A proof engine approach to solving combinational design automation problems. In *Design Automation Conference* (Las Vegas, Nevada, June 2002), pp. 725–730.

[7] ARONS, T., AND PNUELI, A. Verification of data-insensitive circuits: An in-order-retirement case study. In *Conference on Formal Methods in Computer-Aided Design* (Palo Alto, California, Nov. 1998), pp. 351–368.

[8] ARONS, T., AND PNUELI, A. Verifying Tomasulo's algorithm by refinement. In *Conference On VLSI Design* (Goa, India, Jan. 1999), pp. 306–309.

[9] ARONS, T., AND PNUELI, A. Verifying Tomasulo's algorithm by refinement. Tech. Rep. CS98-15, Wiezmann Institute of Science, 1999.

[10] BALL, T., COOK, B., LAHIRI, S. K., AND ZHANG, L. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Conference on Computer-Aided Verification* (Boston, Massachusetts, June 2004), pp. 388–403.

[11] BARRETT, C., AND BEREZIN, S. CVC Lite: A new implementation of the cooperating validity checker. In *Conference on Computer-Aided Verification* (Boston, Massachusetts, June 2004), pp. 515–518.

[12] BARRETT, C., DILL, D., AND LEVITT, J. Validity checking for combinations of theories with equality. In *Conference on Formal Methods in Computer-Aided Design* (Palo Alto, California, Nov. 1996), pp. 187–201.

[13] BEREZIN, S., BIERE, A., CLARKE, E., AND ZHU, Y. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In *Conference on Formal Methods in Computer-Aided Design* (Palo Alto, California, Nov. 1998), pp. 369–386.

[14] BISTRY, D., DULONG, C., GUTMAN, M., JULIER, M., KEITH, M., MENNEMEIR, L. M., MITTAL, M., PELEG, A. D., AND WEISER, U. *The Complete Guide to MMX Technology.* McGraw-Hill, 1997.

[15] BRUCE, K. B., AND MEYER, A. R. The semantics of second order polymorphic lambda calculus. In *Conference on Semantics of Data Types* (Sophia-Antipolis, France, June 1984), pp. 131–144.

[16] BURCH, J. Techniques for verifying superscalar microprocessors. In *Design Automation Conference* (Las Vegas, Nevada, June 1996), pp. 552–557.

[17] BURCH, J., AND DILL, D. Automatic verification of pipelined microprocessor control. In *Conference on Computer-Aided Verification* (Palo Alto, California, June 1994), pp. 60–80.

[18] CASE, B. IA-64's static approach is controversial. *Microprocessor Report 11*, 16 (1997), 22–25.

[19] COOK, B., LAUNCHBURY, J., AND MATTHEWS, J. Specifying superscalar microprocessors with Hawk. In *Workshop on Formal Techniques for Hardware* (Maarstrand, Sweden, June 1998), pp. 155–173.

[20] DAMM, W., AND PNUELI, A. Verifying out-of-order executions. In *Conference on Correct Hardware Design and Verification Methods* (Montreal, Canada, Sept. 1997), pp. 23–47.

[21] DAY, N. A., AAGAARD, M. D., AND COOK, B. Combining stream-based and state-based verification techniques. In *Conference on Formal methods in computer-aided design* (Austin, Texas, Nov. 2000), pp. 126–142.

[22] DULONG, C. The IA-64 architecture at work. *IEEE Computer 31*, 7 (1998), 24–32.

[23] GIRARD, J. Y. The system F of variable types, fifteen years later. *Theoretical Computer Science 2*, 45 (1985), 159–192.

[24] GWENNAP, L. Intel's P6 uses decoupled superscalar design. *Microprocessor Report 9*, 2 (1995), 9–15.

[25] GWENNAP, L. Digital 21264 sets new standard. *Microprocessor Report 14*, 10 (1996), 11–16.

[26] GWENNAP, L. Intel, HP make EPIC disclosure. *Microprocessor Report 11*, 14 (1997), 1–9.

[27] GWENNAP, L. AltiVec vectorizes PowerPC. *Microprocessor Report 12*, 6 (1998), 6–9.

[28] GWENNAP, L. AMD deploys K6-2 with 3DNow. *Microprocessor Report 12*, 7 (1998), 16–17.

[29] GWENNAP, L. Intel outlines high-end roadmap. *Microprocessor Report 12*, 14 (1998), 16–19.

[30] HOSABETTU, R., GOPALAKRISHNAN, G., AND SRIVAS, M. Verifying advanced microarchitectures that support speculation and exceptions. In *Conference on Computer-Aided Verification* (Chicago, Illinois, July 2000), pp. 521–537.

[31] HOSABETTU, R., SRIVAS, M., AND GOPALAKRISHNAN, G. Decomposing the proof of correctness of pipelined microprocessors. In *Conference on Computer-Aided Verification* (Vancouver, Canada, July 1998), pp. 122–134.

[32] HOSABETTU, R., SRIVAS, M., AND GOPALAKRISHNAN, G. Proof of correctness of a processor with reorder buffer using the completion functions approach. In *Conference on Computer-Aided Verification* (Trento, Italy, July 1999), pp. 134–145.

[33] HOSABETTU, R., SRIVAS, M., AND GOPALAKRISHNAN, G. Proof of correctness of a processor without reorder buffer using the completion functions approach. In *Conference on Correct Hardware Design and Verification Methods* (Bad Herrenalb, Germany, Sept. 1999), pp. 8–22.

[34] HOSABETTU, R., SRIVAS, M., AND GOPALAKRISHNAN, G. Formal verification of a complex pipelined processor. *Formal Methods in System Design 23*, 2 (2003), 171–213.

[35] HUCK, J., MORRIS, D., ROSS, J., KNIES, A., MULDER, H., AND ZAHIR, R. Introducing the IA-64 architecture. *IEEE Micro 20*, 5 (2000), 12–23.

[36] HUDAK, P., JONES, S. L. P., AND WADLER, P. Report on the programming language haskell, a non-strict purely functional language (version 1.2). *SIGPLAN Notices 27*, 5 (May 1992), 1–164.

[37] INTEL CORPORATION. SA-110 microprocessor technical reference manual. 2004.

[38] JAGGAR, D. *Advanced RISC Machines Architectural Reference Manual.* Prentice Hall, 1997.

[39] JOHNSON, D. Techniques for mitigating memory latency in the the PA-8500 processor. In *Hot Chips Conference* (Palo Alto, California, Aug. 1998), pp. 145–165.

[40] JONES, M. P. *Qualified Types: Theory and Practice.* PhD thesis, Department of Computer Science, Oxford University, 1992.

[41] JONES, R. B. *Applications Of Symbolic Simulation To The Formal Verification Of Microprocessors.* PhD thesis, Stanford University, Palo Alto, California, 1999.

[42] JONES, R. B. *Symbolic Simulation Methods for Industrial Formal Verification.* Kluwer Academic Publishers, 2002.

[43] JONES, R. B., DILL, D. L., AND BURCH, J. R. Efficient validity checking for processor verification. In *Conference on Computer-Aided Design* (San Jose, California, July 1995), pp. 2–6.

[44] JONES, R. B., SEGER, C.-J. H., AND DILL, D. L. Self-consistency checking. In *Conference on Formal Methods in Computer-Aided Design* (Palo Alto, California, Nov. 1998), pp. 159–171.

[45] JONES, R. B., SKAKKEBAEK, J., AND DILL, D. Reducing manual abstraction in formal verification of out-of-order execution. In *Conference on Formal Methods in Computer-Aided Design* (Palo Alto, California, Nov. 1998), pp. 2–17.

[46] JONES, R. B., SKAKKEBAEK, J., AND DILL, D. Formal verification of out-of-order execution with incremental flushing. *Formal Methods In System Design 2*, 20 (2001), 139–158.

[47] KATHAIL, V., SCHLANSKER, M., AND RAU, B. R. HPL PlayDoh architecture specification: Version 1.0. Tech. Rep. HPL-93-80, Hewlett Packard Laboratories, 1993.

[48] KELLY, E. J., CMELIK, R. F., AND WING, M. J. Memory controller for a microprocessor for detecting a failure of speculation on the physical nature of a component being addressed. United States patent 5832205, Nov. 1998.

[49] MATTHEWS, J., LAUNCHBURY, J., AND COOK, B. Specifying microprocessors in Hawk. In *IEEE Conference on Computer Languages* (Chicago, Illinois, Aug. 1998), pp. 90–101.

[50] MATTHEWS, J. R. *Algebraic Specification and Verification of Processor Microarchitectures.* PhD thesis, Oregon Graduate Institute of Science and Technology, 2000.

[51] MCMILLAN, K. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *Conference on Computer-Aided Verification* (Vancouver, Canada, July 1998), pp. 110–121.

[52] MCMILLAN, K. Circular compositional reasoning about liveness. In *Conference on Correct Hardware Design and Verification Methods* (Bad Herrenalb, Germany, Sept. 1999), pp. 342–345.

[53] MITCHELL, J. C., AND MEYER, A. R. Second-order logical relations. In *Conference on Logics Of Programs* (Brooklyn, New York, Mar. 1985), pp. 225–236.

[54] REYNOLDS, J. C. Types, abstraction, and parametric polymorphism. *Information Processing 1*, 83 (1983), 513–523.

[55] SAWADA, J., AND HUNT, W. Trace table based approach for pipelined microprocessor verification. In *Conference on Computer-Aided Verification* (Haifa, Israel, July 1997), pp. 364–375.

[56] SAWADA, J., AND HUNT, W. Processor verification with precise exceptions and speculative execution. In *Conference on Computer-Aided Verification* (Vancouver, Canada, July 1998), pp. 135–146.

[57] SCHLANSKER, M., RAU, B. R., , MAHLKE, S., KATHAIL, V., JOHNSON, R., ANIK, S., AND ABRAHAM, S. G. Achieving high levels of instruction-level parallelism with reduced hardware complexity. Tech. Rep. HPL-96-120, Hewlett Packard Laboratories, 1996.

[58] SHARANGPANI, H., AND ARORA, K. Itanium processor microarchitecture. *IEEE Micro 20*, 5 (2000), 12–23.

[59] SHRIVER, B., AND SMITH, B. *The Anatomy of a High-Performance Microprocessor: A Systems Perspective.* IEEE Computer Society Press, 1998.

[60] SKAKKEBAEK, J., JONES, R. B., AND DILL, D. Formal verification of out-of-order execution using incremental flushing. In *Conference on Computer-Aided Verification* (Vancouver, Canada, July 1998), pp. 98–109.

[61] SONG, P. Demystifying EPIC and IA-64. *Microprocessor Report 12*, 1 (1998), 21–27.

[62] TULLSEN, D. M., EGGERS, S. J., EMER, J. S., LEVY, H. M., LO, J. L., AND STAMM, R. L. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *International Symposium on Computer Architecture* (Philadelphia, Pennsylvania, May 1996), pp. 191–202.

[63] VELEV, M. N., AND BRYANT, R. E. Bit-level abstraction in the verification of pipelined microprocessors by correspondence checking. In *Conference on Formal Methods in Computer-Aided Design* (Palo Alto, California, Nov. 1998), pp. 18–35.

[64] VELEV, M. N., AND BRYANT, R. E. Superscaler processor verification using efficient reductions of the logic of equality with uninterpreted functions to propositional logic. In *Conference on Correct Hardware Design and Verification Methods* (Bad Herrenalb, Germany, Sept. 1999), pp. 37–53.

[65] VELEV, M. N., AND BRYANT, R. E. Formal verification of superscaler microprocessors with multicycle functional units, exceptions, and branch prediction. In *Design Automation Conference* (Los Angeles, California, June 2000), pp. 112–117.

[66] WADLER, P. Theorems for free! In *Symposium on Functional Programming and Computer Architecture* (London, United Kingdom, Sept. 1989), pp. 347–359.

Biographical Note

(John) Byron Cook was born in Colorado on the 15th of March, 1971. He received his B.Sci. degree from The Evergreen State College in 1995. While still a Ph.D. student Byron interned at Intel's Strategic CAD laboratories in Hillsboro, Oregon. In 2000, Byron joined Prover Technology AB in Stockholm, Sweden, where he continued his Ph.D. research part-time. He later joined Microsoft Research in 2002, where he is now searching for new algorithms and techniques for the formal verification of software. Byron lives in Cambridge, England.