# Operation-Centric Hardware Description and Synthesis

by

James C. Hoe

B.S., University of California at Berkeley (1992)
M.S., Massachusetts Institute of Technology (1994)

Submitted to
the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2000

© Massachusetts Institute of Technology 2000. All rights reserved.

Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
April 28, 2000

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Arvind
Johnson Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

# Operation-Centric Hardware Description and Synthesis

by

James C. Hoe

Submitted to the
Department of Electrical Engineering and Computer Science
on April 28, 2000, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

## Abstract

In an operation-centric framework, the behavior of a system is decomposed and described as a collection of operations. An operation is defined by a predicate condition and an effect on the system's state. An execution of the system corresponds to some sequential interleaving of the operations such that each operation in the sequence produces a state that enables the next operation. An operation's effect on the system is global and atomic. In other words, an operation "reads" the state of the system in one step, and, if enabled, the operation updates the state in the same step. This atomic semantics simplifies the task of hardware description by permitting the designer to formulate each operation as if the system were otherwise static.

This thesis develops a method for hardware synthesis from an operation-centric description. The crux of the synthesis problem is in finding a synchronous state transition system that carries out multiple parallelizable operations per clock cycle and yet maintains a behavior that is consistent with the atomic and sequential semantics of the operations. The thesis first defines an Abstract Transition System (ATS), an operation-centric state machine abstraction. The thesis next develops the theories and algorithms to synthesize an efficient synchronous digital circuit implementation of an ATS. Finally, the thesis presents TRSpec, a source-level operation-centric hardware description language based on the formalism of Term Rewriting Systems.

The results of this thesis show that an operation-centric framework offers a significant improvement over traditional hardware design flows. The TRSpec language and synthesis algorithms developed in this thesis have been realized in the Term Rewriting Architectural Compiler (TRAC). This thesis presents the results of several operation-centric design exercises using TRSpec and TRAC. In an example based on a 32-bit MIPS integer core, the operation-centric description can be developed five times faster than a hand-coded structural Verilog description. Nevertheless, the circuit implementation produced by the operation-centric framework is comparable to the hand-coded design in terms of speed and area.

Thesis Supervisor: Arvind
Title: Johnson Professor of Computer Science and Engineering

# Acknowledgments

First and foremost, I would like to thank Professor Arvind for supervising this research. His insight and criticism added greatly to this thesis. Furthermore, his guidance and support have been invaluable to my academic development over the last eight years. I wish him the best of luck in the Sandburst venture.

I want to thank Professor Srinivas Devadas and Professor Martin Rinard for generously spending their time to advise me on this thesis. They have been both my mentors and friends.

My fellow *dataflowers* from the Computation Structures Group (CSG) have helped me in so many ways. I want to thank Dr. Larry Rudolph who has always been a source of good ideas. I will always treasure the company of Shail Aditya, Andy Shaw, Boon Ang, Alejandro Caro, Derek Chiou, Xiaowei Shen, Jan Maessen, Mike Ehrlich and Daniel Rosenband,[1] my long-time fellow CSG students. Daniel Rosenband provided the hand-written Verilog descriptions in this thesis and generated many of the synthesis results. Mieszko Lis reviewed many drafts of this thesis. I also want to thank Dr. Andy Boughton for looking after everyone in the group. CSG and everyone associated with it have been a big part of my life for nearly a decade. I hope that, one day, I will finally come to know what exactly is a "computation structure".

I want to thank the Intel Corporation and the Intel Foundation for supporting the research in this thesis by providing research grants and a fellowship. I also appreciate the many useful exchanges with Dr. Timothy Kam and other members of Intel's Strategic CAD Laboratories.

I owe much to my family for affording me the luxury of "just hanging around" as a graduate student for so many years. As always, I am indebted to my parents for their love, support, and encouragement through the years. I am grateful for their hard work and sacrifices that have allowed me to be who I am today. Last but not the least, I must thank my infinitely patient wife for her love and support during my studies at MIT. She encourages me when I am down, understands me when I am upset, and tolerates all of my many quirks. As my most loyal reader, she has tirelessly proofread every document I have produced at MIT.

I would like to finish by telling a short story with me in it. It was the summer of 1987 in Pasadena, California. I was attending an informal lecture given by Richard Feynman to a class of SSSSP[2] high school students. Near the end of the lecture, he challenged us with a puzzle to find the flaw in an odd but apparently sound explanation for universal gravitation. As soon as the lecture ended, I charged to the front of the room to check my answer with him. Instead of agreeing with me, he gave me a knowing smirk and put both of his hands on my head. To my surprise, he then grabbed my head, as one would with a basketball, and proceeded to shake it in comic exaggeration. Dumbfounded, I heard him say in a playful voice, **"Don't just ask me. Use your head to think about it. That is what it's there for!"** At that instant, I realized the point in learning is not so one can come up with the

[1]Names appear in the chronological order of their realized or pending graduation dates.

[2]Caltech's Summer Secondary School Science Project

same answers as the answer key on a final exam. What could be the point in solving a cooked up problem that has been solved hundreds of times by hundreds of people? I acquired a whole new mindset where the end result of learning is not rote knowledge but the ability to solve real-world problems that have not been conveniently prepared from their right answers. For me, finding answers to open questions is not only the true point in learning, but it is also where the true act of learning begins. Finding things out is indeed pleasurable [Fey99]. Incidentally, for those of you who are wondering, I did come up with the right answer on that day.

*In memory of my grandfather,*

*Hoe Kwan-Wu, M.D., 1900 ∼ 1996*

# Contents

# List of Figures

# Chapter 1

# Introduction

This thesis presents a method for hardware synthesis from an "operation centric" description. In an operation-centric description, the behavior of a system is decomposed and described as a collection of operations. This operation-centric view of hardware is new in synthesis but not in description. Most high-level hardware specifications intended for human reading are given operation-centrically. A typical assembly programmer's manual for a microprocessor is an example where the behavior of the processor is broken down into per-instruction operations. Although informal, the written material in most computer architecture textbooks also presents ideas and designs operation-centrically. This thesis improves the usefulness of the operation-centric approach to hardware description by developing a formal description framework and by enabling automatic synthesis of an efficient synchronous circuit implementation. The results of this thesis show that an operation-centric hardware development framework offers a significant reduction in design time and effort, without loss in implementation quality, when compared to traditional frameworks.

## 1.1 Operation-Centric Hardware Description

An operation is defined by a predicate condition and an effect. The interpretation is that an operation's effect on the system state can take place when the predicate condition is satisfied. Although an implementation may execute multiple operations concurrently, the end result of an execution must correspond to some sequential interleaving of the operations such that each operation in the sequence produces a state that enables the next operation.

For an unambiguous interpretation, the effect of an operation is taken to be atomic. In other words, an operation "reads" the entire state of the system in one step, and, if the operation is enabled, the operation updates the state in the same step. If several operations are enabled in a state, any one of the operations can be selected to update the state in one step, and afterwards a new step begins with the updated state. This atomic semantics simplifies the task of hardware description by permitting the designer to formulate each operation as if the system were otherwise static. The designer does not have to worry about unexpected interactions with other concurrent operations. For related reasons, the atomic semantics of operations also makes an operational-centric description easier to interpret by a human. It is important to re-emphasize that this sequential and atomic interpretation of a description does not prevent a legal implementation from executing several operations concurrently, provided the concurrent execution does not introduce new behaviors that are not producible by sequential executions.

The instruction reorder buffer (ROB) of a modern out-of-order microprocessor exemplifies complexity and concurrency in hardware behavior.[1] Describing an ROB poses a great challenge for traditional hardware description frameworks where concurrency needs to be managed explicitly. However, in an operation-centric framework, the behavior of an ROB can be perspicuously described as a collection of atomic operations including *dispatch*, *complete*, *commit*, etc. [AS99]. For example, the *dispatch* operation is specified to take place if there is an instruction that has all of its operands and is waiting to execute, and, furthermore, the execution unit needed by the instruction is available. The effect of the *dispatch* operation is to send the instruction to the execution unit. The specification of the *dispatch* operation need not include information about how to resolve potential conflicts arising from the concurrent execution of other operations.

## 1.2 Limitations of Operation-Centric Frameworks

Instead of marking the progress of time with an explicit global clock, the behavior of an operation-centric description is interpreted as a sequence of operations executed in atomic steps. This abstract model of time permits an unambiguous sequential interpretation by the designer but at the same time enables a compiler to automatically exploit parallelism by scheduling multiple parallelizable operations into the same clock cycle in a synchronous implementation. Unfortunately, at times, this normally simplifying abstraction gets in the way of describing hardware designs whose correctness criteria include a specific synchronous execution timing. For example, a microprocessor often has to interface with the memory controller via a synchronous bus protocol that assumes predefined latencies between the exchanges of bus signals. As another example, suppose the description of the out-of-order processor above contains a fully-pipelined multiplier unit with a latency of exactly four clock cycles. After a *dispatch* operation launches a multiply instruction, a corresponding *complete* operation should be triggered exactly four cycles later. This kind of temporal relationships between operations cannot be expressed directly in an operation-centric framework. In more severe cases, glue logic expressed in a lower-level synchronous representation becomes necessary. However, in many cases, this limitation can be avoided by using an asynchronous handshake between the interacting elements. For example, an operation-centrically described processor can interface with a bus protocol where the initiations of exchanges are demarcated by appropriate strobe signals. The multiplication unit in the out-of-order processor can be outfitted with *ready* and *done* status

[1]Refer to [HP96] and [Joh91] for background information on the operation of an ROB.

signals that can be tested by the predicates of the *dispatch* and *complete* operations.

## 1.3 Comparison to CFSM-based Frameworks

Digital hardware designs inherently embody highly concurrent behaviors. The implementation of any non-trivial design invariably consists of a collection of cooperating finite state machines (CFSM). Hence, most hardware description frameworks, whether schematic or textual, use CFSM as the underlying abstraction. In a CFSM framework, a designer explicitly manages concurrency by scheduling the exact cycle-by-cycle interactions between multiple concurrent state machines. It is easy to make a mistake in coordinating interactions between two state machines because transitions in different state machines are not coupled semantically. It is also difficult to modify one state machine without considering its interaction with the rest of the system.

The advantage of a CFSM framework lies in its resemblance to the underlying circuit implementation. This makes automatic circuit synthesis simpler and more efficient. The close correlation between description and implementation also gives designers tighter control over lower-level implementation choices. On the other hand, a CFSM description cannot be easily correlated to the functionality of the design. Producing a CFSM-based description from a high-level design specification requires considerable time and expertise because the designer has to translate between two very different abstractions. The development of a CFSM-based description is also error-prone due to its explicit concurrency model. The disadvantages of CFSM frameworks can quickly outweigh their advantages as hardware design complexity increases.

## 1.4 Comparison to Other High-Level Frameworks

**RTL Description and Synthesis:** Synthesizing hardware from textual register-transfer level (RTL) descriptions is currently the standard practice in the development of digital integrated circuits. Many high-end integrated-circuit companies own proprietary hardware description languages (HDLs) and synthesis tools to cater to their applications and fabrication technologies. On the other hand, Verilog [TM96] and VHDL[2] [Ins88] are two standardized HDLs that are supported by commercial tools and are widely used in both industry and academia. The Synopsis Design Compiler [Synb] and Synplify [Synd] are commercial synthesis tools that compile the RTL (a.k.a. structural) subsets of Verilog and VHDL. Synthesizable RTL descriptions rely on the same synchronous CFSM abstraction as traditional schematic frameworks. In comparison to schematic capture, the productivity gain of an RTL-based design flow comes from the relative compactness of textual descriptions and from automatic logic optimizations by synthesis tools. The operation-centric synthesis framework in this thesis outputs RTL descriptions and relies on commercial hardware compilers to produce the final circuit implementations.

[2]VHSIC Hardware Description Language, where VHSIC stands for Very High Speed Integrated Circuits.

**Behavioral Description and Synthesis:** The term *behavioral description* typically refers to describing hardware, or hardware/software systems, as multiple threads of computation that communicate via message passing or a shared-memory paradigm. The underlying multithreaded abstraction can be traced to Communicating Sequential Processes [Hoa85]. The objective of behavioral synthesis is to infer and allocate the necessary hardware resources and schedule their usage by the threads. Topics relating to this style of high-level synthesis are discussed in [GDWL92] and [MLD92]. The behavioral frameworks describe each hardware thread using high-level sequential languages like C or behavioral Verilog. A comprehensive summary of early efforts can be found in [WC91] and [CW91]. More recent efforts are represented by HardwareC [Sta90], Esterel [Ber98], ECL [LS99], SpecC [GZD$^+$00], and a hybrid C/Verilog co-specification framework by Thomas, et al. [TPPW99]. Commercially, besides behavioral Verilog and VHDL, SystemC [LTG97] and Superlog [Co-], based on C/C++ and C/Verilog respectively, are two threaded behavioral languages that are currently gaining acceptance. The multithreaded behavioral abstraction is an improvement over the CFSM frameworks. Instead of synchronizing cycle-by-cycle, the threaded abstraction allows hardware threads to synchronize at coarser granularity or only at communication points. Nevertheless, a designer still needs to coordinate the interactions between concurrent threads of computations explicitly.

**Software Languages for Hardware Description and Synthesis:** Both sequential and parallel programming languages have been used to capture functionalities for hardware implementation. Transmogrifier-C is based on C plus additional hardware-specific constructs [Gal95]. The Programmable Active Memory (PAM) project uses C++ syntax for RTL description [VBR$^+$96]. These first two usages of software languages are strongly influenced by hardware description needs. Although these languages leverage the familiar syntax of C and C++, a description is typically illegal or not meaningful as a C or C++ program. Semantically and syntactically correct sequential C and Fortran programs have been automatically parallelized to target an array of configurable structures in the RAW project [BRM$^+$99]. The SpC project attempts to synthesize full ANSI-compliant C programs and, in particular, addresses C programs with pointers to variables [SM98, Mic99]. Data-parallel C languages have also been used to program an array of FPGAs in Splash 2 [GM93] and CLAy [GG97]. These latter examples synthesize hardware from semantically correct programs and thus are convenient tools for implementing algorithms. However, these program-based descriptions are not suitable for describing microarchitectural mechanisms. In the operation-centric framework of this thesis, both algorithms and microarchitectures can be expressed in the TRSpec language for synthesis.

**Processor-Specific High-level Description and Synthesis:** High-level hardware description and synthesis are employed in the development of application specific instruction set processors (ASIPs). For example, the ADAS [PSH$^+$92] environment accepts an instruction set architecture (ISA) described in Prolog and emits a VLSI implementation using a combination of tools in stages. During behavioral synthesis, the Piper tool attempts to pipeline the microarchitecture while taking into account factors like instruction issue frequencies, pipeline stage latencies, etc. The whole ADAS development suite is driven at the front-end by ASIA [HHD93], a system that automatically generates a custom ISA for a particular application program. Other processor-specific high-level description and synthesis frameworks include Mimola [Mar84], Dagar [Raj89], nML [FPF95], and ISDL [HHD97]. Although not domain-specific to ASIP developments, the operation-centric framework in this thesis can also be used to give concise specifications of ISA semantics and processor microarchitectures. Furthermore, an operation-centric ISA specification is amenable to automatic synthesis and architectural transformations.

**Other Representative Work:** Hardware description frameworks with formal basis have been used in the context of formal design specification and verification. For example, the specification language of the HOL theorem proving system [SRI97] has been used to describe a pipelined processor, and a methodology has been developed to verify a pipelined processor description against its non-pipelined counterpart [Win95]. The Hawk language, based on Haskell [JHA$^+$98], can be used to create executable specifications of processor microarchitectures [MLC98]; a Hawk pipelined processor specification can be reduced into a non-pipelined equivalent for verification [ML99]. The TRSPEC language described in this thesis is based on the formalism of Term Rewriting Systems (TRS). Besides being synthesizable, a TRSPEC description is also amenable to formal verification. In a closely related research, Marinescu and Rinard address the problem of synthesizing a synchronous pipeline from a description of loosely-coupled modules connected by queues [MR99].

## 1.5 Thesis Contributions

The operation-centric view of hardware has existed in many forms of hardware specifications, usually to convey high-level architectural concepts. This thesis creates a new hardware development framework where an efficient circuit implementation can be synthesized automatically from an operation-centric description. Specifically, this thesis makes the following contributions:

- Identification of key properties and advantages of operation-centric hardware description frameworks
- Design of TRSPEC, a source-level operation-centric hardware description language based on the TRS formalism
- Definition of an operation-centric state machine abstraction that serves as the syntax-independent intermediate representation during hardware synthesis
- Formulation of the theories and algorithms necessary to create an efficient hardware implementation from an operation-centric description

- Implementation of the Term Rewriting Architectural Compiler (TRAC), a compiler for TRSpec
- Investigation of the effectiveness of TRSpec and TRAC in comparison to traditional RTL-based development flows
- Preliminary investigation of pipelined superscalar processor development via source-to-source transformations of TRSpec descriptions

In this early step towards the study of operation-centric hardware development frameworks, several important and interesting points of research are not fully addressed. These open issues are summarized in Section 7.2

## 1.6 Thesis Organization

Following this introductory chapter, the next chapter first provides a further introduction to operation-centric hardware descriptions using four design examples. Chapter 3 then explains the synthesis of operation-centric hardware descriptions. The chapter first develops a formal representation for operation-centric hardware description and then describes the synthesis algorithms implemented in TRAC. Chapter 4 presents TRSpec, the source-level TRS language accepted by TRAC. Chapter 5 presents the results from applying TRAC to TRSpec descriptions. Chapter 6 presents the application of TRSpec and TRAC to the design of a pipelined superscalar processor. Chapter 7 concludes with a summary of this thesis and identifies areas for continued investigation.

# Chapter 2

# Operation-Centric Design Examples

Four design examples are explored to provide a concrete demonstration of operation-centric hardware description. The first example develops a hardware implementation of Euclid's Algorithm. The second example describes a simple instruction set architecture (ISA). The third and fourth examples describe two different pipelined implementations of the same ISA. In this chapter, the operation-centric descriptions borrow the notation of Term Rewriting Systems (TRS) [BN98, Klo92].

## 2.1 Euclid's Algorithm

This example develops a finite state machine (FSM) that computes the greatest common divisor (GCD) of two positive integers using Euclid's Algorithm.[1] The FSM's state elements consist of two 32-bit registers, $a$ and $b$. The behavior of this FSM can be captured operation-centrically using a TRS whose terms has the signature Gcd(a,b) where the variables a and b correspond to the unsigned integer contents of registers $a$ and $b$. The FSM has two operations, *Mod* and *Flip*. The *Mod* operation can be described by the rewrite rule:

*Mod Rule:*
$\mathsf{Gcd(a,b)\ if\ (a{\geq}b) \wedge (b{\neq}0)} \rightarrow \mathsf{Gcd(a\text{-}b,b)}$

In general, a rewrite rule has three components: a left-hand-side pattern term, a right-hand-side rewrite term, and an optional predicate. In the *Mod* rule, the pattern is Gcd(a,b), the rewrite term is Gcd(a-b,b), and the predicate is the expression $\mathsf{(a{\geq}b) \wedge (b{\neq}0)}$. A rule can be applied if the pattern matches the current state value and if the predicate is true. In the context of operations, the pattern and the predicate describe the *condition* of an operation, i.e., the *Mod* operation is enabled only

[1]Euclid's Algorithm states $Gcd(a,b) = Gcd(b,(a\ mod\ b))\ if\ b \neq 0$. This equality can be used to repeatedly reduce the problem of computing $Gcd(a,b)$ to a simpler problem. For example, $Gcd(2,4) = Gcd(4,2) = Gcd(2,0) = 2$.

when $(a \geq b) \wedge (b \neq 0)$. The right-hand-side rewrite term specifies the new state value after a rewrite. In describing the behavior of an FSM, the rewrite term conveys how an operation updates the state elements. The effect of the *Mod* operation is to update the register $a$ by the value $a - b$.

The *Flip* operation can be described by the rewrite rule:

*Flip Rule:*
Gcd(a,b) if (a<b) $\rightarrow$ Gcd(b,a)

The effect of the *Flip* operation is to exchange the contents of registers $a$ and $b$. This operation is enabled only when $(a < b)$.

Starting with two integers in registers $a$ and $b$, an execution of the FSM computes the GCD of these initial values. The execution of the FSM must correspond to some sequential and atomic interleaving of the *Mod* operation and the *Flip* operation. The sequence of operations stops when register $b$ contains 0 because neither operation can be enabled. The GCD of the initial values is in register $a$ at the end of the sequence. For example, starting with an initial state corresponding to the term Gcd(2,4), the sequence of operations that follows is

$$\mathsf{Gcd(2,4)} \rightarrow^{Flip} \mathsf{Gcd(4,2)} \rightarrow^{Mod} \mathsf{Gcd(2,2)} \rightarrow^{Mod} \mathsf{Gcd(0,2)} \rightarrow^{Flip} \mathsf{Gcd(2,0)}$$

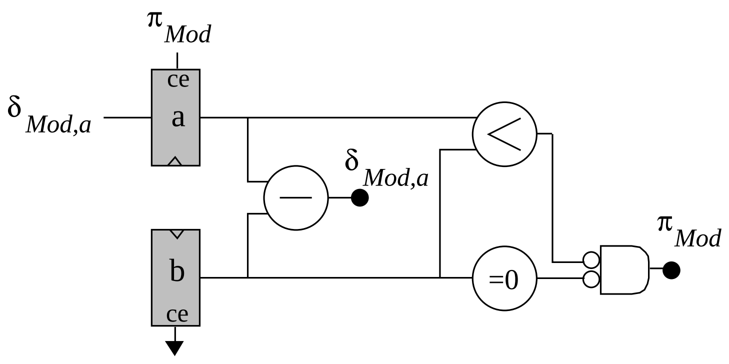
This operation-centric description of an FSM is not only easy to understand, but it also allows certain properties of the whole system to be proven by examining the property of each operation individually. For example, it can be shown that if an FSM correctly implements this two-operation system, then it maintains the invariant that the GCD of the contents of registers $a$ and $b$ never changes. To show this, one can make the argument that neither the *Mod* operation nor the *Flip* operation can change the GCD of $a$ and $b$.[2] Since an execution of a correctly implemented FSM must correspond to some sequential and atomic interleaving of these two operations, the GCD of $a$ and $b$ should never change from operation to operation.

In a TRS, the state of the system is represented by a collection of values, and a rule rewrites values to values. Given a collective state value $\mathsf{s}$, a TRS rule computes a new value $\mathsf{s}'$ such that

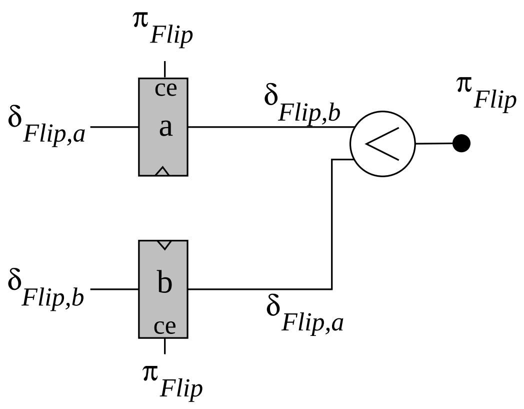
$$\mathsf{s}' = \textit{if } \pi(\mathsf{s}) \textit{ then } \delta(\mathsf{s}) \textit{ else } \mathsf{s}$$

where the $\pi$ function captures the firing condition and the $\delta$ function captures the effect of the rule. It is also possible to view a rule as a state-transformer in a state-based system. In the state-transformer view, the applicability of a rule is determined by computing the $\pi$ function on the current state, and the next-state logic consists of a set of actions that alter the contents of the state elements to match the value

[2]For the *Mod* operation, if $(a \geq b) \wedge (b \neq 0)$ then $(a \textit{ mod } b) = ((a - b) \textit{ mod } b)$, and thus, $Gcd(a, b) = Gcd(a \textit{ mod } b, b) = Gcd((a - b) \textit{ mod } b, b) = Gcd(a - b, b)$. For the *Flip* operation, $Gcd(a, b) = Gcd(b, a)$.



(a)



(b)

Figure 2-1: Next-state logic corresponding to (a) the *Mod* operation and (b) the *Flip* operation.

of $\delta(s)$. The execution semantics of operations remains sequential and atomic in this action-on-state interpretation.

The operations in this example can be restated in terms of actions. In the notation below, $\pi_{Mod}$ is the predicate of the *Mod* operation, $\mathbf{a}_{Mod,a}$ is the action on register $a$ according to the *Mod* operation, etc.

$$\begin{aligned} \pi_{Mod} &= (a \geq b) \wedge (b \neq 0) \\ \mathbf{a}_{Mod,a} &= \boldsymbol{set}(a - b) \end{aligned}$$

$$\begin{aligned} \pi_{Flip} &= (a < b) \\ \mathbf{a}_{Flip,a} &= \boldsymbol{set}(b) \\ \mathbf{a}_{Flip,b} &= \boldsymbol{set}(a) \end{aligned}$$

If an operation is mapped to a synchronous state transition of the FSM, $\pi_{Mod}$ and $\mathbf{a}_{Mod,a}$ map to the next-state logic in circuit (a) of Figure 2-1. The *Flip* operation maps to the next-state logic in circuit (b) of Figure 2-1.

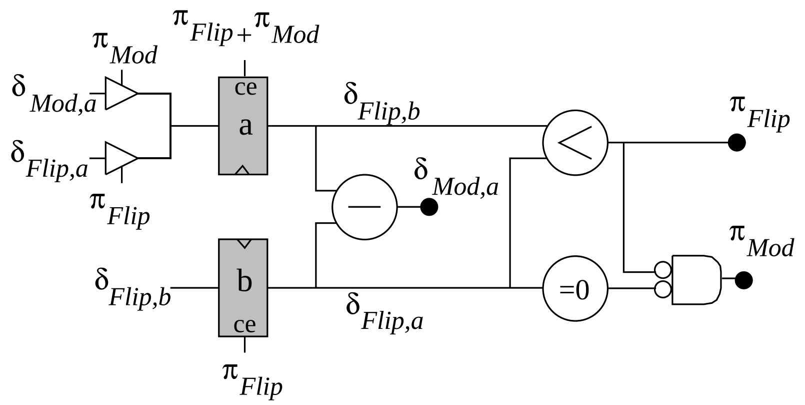


Figure 2-2: Circuit for computing $Gcd(a, b)$.

The final FSM implementation can be derived by combining the two circuits from Figure 2-1. Both the *Mod* operation and the *Flip* operation affect register $a$, but the two operations are never enabled in the same state. (Their predicates contradict.) Since the two operations are mutually exclusive, the latch enable for $a$ is simply the logical-$OR$ of the two operations' predicates (i.e., $\pi_{Mod} + \pi_{Flip}$) and the next-state value of $a$ is selected from either $\mathbf{a}_{Mod,a}$ or $\mathbf{a}_{Flip,a}$ through a multiplexer. The next-state value and latch enable of $b$ is controlled by the *Flip* operation alone.

Figure 2-2 shows the merged circuit. The circuit behaves like circuit (a) of Figure 2-1 in a cycle when the condition $(a \geq b) \wedge (b \neq 0)$, required by the *Mod* operation, is satisfied. In a cycle when the condition $(a < b)$, required by the *Flip* operation, is satisfied, the circuit behaves like circuit (b) of Figure 2-1 instead. Since the two operations are mutually exclusive, the atomic semantics of operations is maintained in this implementation automatically.

## 2.2 A Simple Processor

A simple ISA can be specified operation-centrically as a TRS whose terms have the signature Proc(pc, rf, imem, dmem). The four fields of a processor term correspond to the values of the programmer-visible states of the ISA. The variable pc represents the program counter register. The rf variable corresponds to the register file, an array of integer values. In an expression, rf[r] gives the value stored in location r of rf, and rf[r:=v] gives the new value of the array after location r has been updated by the value v. The imem variable is an array of instructions. The dmem variable corresponds to the data memory, another array of integer values.

The behavior of the processor can be described by giving a rewrite rule for each instruction in the ISA. The following rule describes the execution of an *Add* instruction.

*Add Rule:*

    Proc(pc, rf, imem, dmem) where Add(rd,r1,r2)=imem[pc]
$\rightarrow$ Proc(pc+1, rf[rd:=(rf[r1]+rf[r2])], imem, dmem)

This rule conveys the same level of information as one would find in a programmer's manual. The rule is enabled for a processor term whose current instruction, imem[pc], is an *Add* instruction. (An *Add* instruction is represented by a term with the signature Add(rd,r1,r2). The rule uses a pattern-matching *where* statement to require imem[pc] to match the pattern Add(rd,r1,r2).) Given a processor term that satisfies this condition, the rule's rewrite term specifies that the processor term's pc field should be incremented by 1 and the destination register rd in rf should be updated with the sum of rf[r1] and rf[r2]. In the state-transformer view, the operation described by the *Add* rule can be stated as

$$\begin{aligned}
\pi_{Add} &= (\mathsf{imem[pc]{=}Add(rd,r1,r2)}) \\
\mathbf{a}_{Add,\mathsf{pc}} &= \boldsymbol{set}(\mathsf{pc{+}1}) \\
\mathbf{a}_{Add,\mathsf{rf}} &= \boldsymbol{a\text{-}set}(\mathsf{rd,rf[r1]{+}rf[r2]})
\end{aligned}$$

Figure 2-3 gives the rewrite rules for all instructions in the ISA. The instructions are move PC to register, load immediate, register-to-register addition and subtraction, branch if zero, memory load and store. The complete TRS not only describes an ISA, but it also constitutes an operation-centric description of an FSM that implements the ISA. The datapath for the processor can be derived automatically and is depicted in Figure 2-4.

## 2.3 A Pipelined Processor

The ISA description from the previous example can be transformed into the description of a pipelined processor by adding FIFOs as pipeline buffers and by systematically splitting every instruction operation into sub-operations local to the pipeline stages.

For example, a two-stage pipelined processor can be specified as a TRS whose terms have the signature Proc(pc, rf, bf, imem, dmem). A FIFO bf is included as the pipeline buffer between the fetch stage and the execute stage. The FIFO is abstracted to have a finite but unspecified size. Using a list-like syntax, a FIFO containing three elements can be expressed as b1;b2;b3. The two main operations on a FIFO are enqueue and dequeue. Enqueuing b to bf yields bf;b while dequeuing from b;bf leaves bf. An empty FIFO is expressed as '$\epsilon$'.

Using a FIFO to separate pipeline stages provides the necessary isolation that allows the operations in one stage to be described independently of the other stages. Although this style of pipeline description reflects a pipeline that is asynchronous and elastic, it is possible to infer a legal implementation that is fully-synchronous and has stages separated by simple registers.

The *Add* rule from the previous example can be split into two sub-rules that describe the operations in the fetch stage and the execute stage of the pipeline. The fetch-stage sub-rule is

*Fetch Rule:*

```
     Proc(pc, rf, bf, imem, dmem)
→    Proc(pc+1, rf, bf;imem[pc], imem, dmem)
```

```
Loadi Rule:
      Proc(pc, rf, imem, dmem) where Loadi(rd,const)=imem[pc]
→     Proc(pc+1, rf[rd:=const], imem, dmem)

Loadpc Rule:
      Proc(pc, rf, imem, dmem) where Loadpc(rd)=imem[pc]
→     Proc(pc+1, rf[rd:=pc], imem, dmem)

Add Rule:
      Proc(pc, rf, imem, dmem) where Add(rd,r1,r2)=imem[pc]
→     Proc(pc+1, rf[rd:=(rf[r1]+rf[r2])], imem, dmem)

Sub Rule:
      Proc(pc, rf, imem, dmem) where Sub(rd,r1,r2)=imem[pc]
→     Proc(pc+1, rf[rd:=(rf[r1]-rf[r2])], imem, dmem)

Bz Taken Rule:
      Proc(pc, rf, imem, dmem) if (rf[rc]=0) where Bz(rc,ra)=imem[pc]
→     Proc(rf[ra], rf, imem, dmem)

Bz Not-Taken Rule:
      Proc(pc, rf, imem, dmem) if (rf[rc]≠0) where Bz(rc,ra)=imem[pc]
→     Proc(pc+1, rf, imem, dmem)

Load Rule:
      Proc(pc, rf, imem, dmem) where Load(rd,ra)=imem[pc]
→     Proc(pc+1, rf[rd:=dmem[rf[ra]]], imem, dmem)

Store Rule:
      Proc(pc, rf, imem, dmem) where Store(ra,r)=imem[pc]
→     Proc(pc+1, rf, imem, dmem[rf[ra]:=rf[r]])
```

Figure 2-3: TRS rules for a simple ISA.

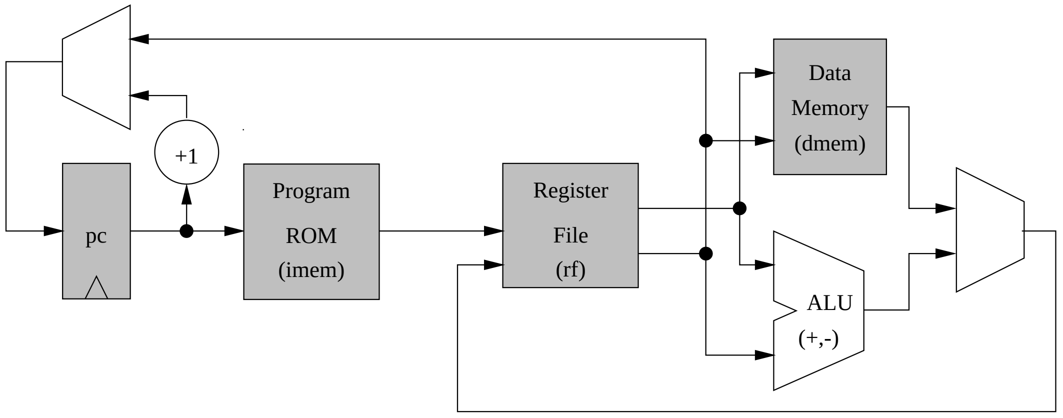


Figure 2-4: A simple non-pipelined processor datapath, shown without control signals.

The execute-stage sub-rule is

*Add Execute Rule:*
    Proc(pc, rf, inst;bf, imem, dmem) where Add(rd,r1,r2)=inst
$\rightarrow$ Proc(pc, rf[rd:=(rf[r1]+rf[r2])], bf, imem, dmem)

The *Fetch* rule fetches instructions from consecutive instruction memory locations and enqueues them into bf. The *Fetch* rule is not concerned with what happens if a branch is taken or if the pipeline encounters an exception. The *Add Execute* rule, on the other hand, would process the next pending instruction in bf as long as it is an *Add* instruction.

In this TRS, more than one rule can be enabled on a given state. The *Fetch* rule is always ready to fire, and at the same time, the *Add Execute* rule, or other execute-stage rules, may be ready to fire as well. Even though conceptually only one rule should be fired in each step, an implementation of this processor description must carry out the effect of both rules in the same clock cycle. Without concurrent execution, the implementation does not behave like a pipeline. Nevertheless, the implementation must also ensure that a concurrent execution of multiple operations produces the same result as a sequential execution.

The *Bz Taken* rule and the *Bz Not-Taken* rule from the previous example can also be split into separate fetch and execute rules. The execute-stage components of the *Bz Taken* and *Bz Not-Taken* rules are

*Bz Taken Execute Rule:*
    Proc(pc, rf, inst;bf, imem, dmem) if (rf[rc]=0) where Bz(rc,ra)=inst
$\rightarrow$ Proc(rf[ra], rf, $\epsilon$, imem, dmem)

and

*Bz Not-Taken Execute Rule:*
    Proc(pc, rf, inst;bf, imem, dmem) if (rf[rc]$\neq$0) where Bz(rc,ra)=inst
$\rightarrow$ Proc(pc, rf, bf, imem, dmem)

The *Fetch* rule performs a weak form of branch speculation by always incrementing pc. Consequently, in the execute stage, if a branch is resolved to be taken, besides setting pc to the branch target, all speculatively fetched instructions in bf need to be discarded. This is indicated by setting bf to '$\epsilon$' in the *Bz Taken Execute* rule.

Although both the *Fetch* rule and the *Bz Taken Execute* rule can affect pc and bf, the sequential semantics of rules allows the formulation of the *Bz Taken Execute* rule to ignore the possibility of contentions or race conditions with the *Fetch* rule. In a clock cycle where the processor state enables both rules, the description permits an implementation to behave as if the two operations are executed sequentially, in either order. This implementation choice determines whether one or two pipeline bubbles are inserted after a taken branch, but it does not affect the processor's ability to correctly execute a program.

The operations in this example can also be restated in terms of actions:

$$\begin{aligned}
\pi_{Fetch} &= \textbf{\textit{notfull}}(\mathsf{bf}) \\
\mathbf{a}_{Fetch,\mathsf{pc}} &= \textbf{\textit{set}}(\mathsf{pc}{+}1) \\
\mathbf{a}_{Fetch,\mathsf{bf}} &= \textbf{\textit{enq}}(\mathsf{imem}[\mathsf{pc}])
\end{aligned}$$

$$\begin{aligned}
\pi_{Add-Exec} &= (\textbf{\textit{first}}(\mathsf{bf}){=}\mathsf{Add}(\mathsf{rd,r1,r2}))\wedge\textbf{\textit{notempty}}(\mathsf{bf}) \\
\mathbf{a}_{Add-Exec,\mathsf{rf}} &= \textbf{\textit{a-set}}(\mathsf{rd,rf[r1]{+}rf[r2]}) \\
\mathbf{a}_{Add-Exec,\mathsf{bf}} &= \textbf{\textit{deq}}(\,)
\end{aligned}$$

$$\begin{aligned}
\pi_{BzTaken-Exec} &= (\textbf{\textit{first}}(\mathsf{bf}){=}\mathsf{Bz}(\mathsf{rc,rt}))\wedge(\mathsf{rf[rc]}{=}0)\wedge\textbf{\textit{notempty}}(\mathsf{bf}) \\
\mathbf{a}_{BzTaken-Exec,\mathsf{pc}} &= \textbf{\textit{set}}(\mathsf{rf[ra]}) \\
\mathbf{a}_{BzTaken-Exec,\mathsf{bf}} &= \textbf{\textit{clear}}(\,)
\end{aligned}$$

$$\begin{aligned}
\pi_{BzNotTaken-Exec} &= (\textbf{\textit{first}}(\mathsf{bf}){=}\mathsf{Bz}(\mathsf{rc,rt}))\wedge(\mathsf{rf[rc]}{\neq}0)\wedge\textbf{\textit{notempty}}(\mathsf{bf}) \\
\mathbf{a}_{BzNotTaken-Exec,\mathsf{bf}} &= \textbf{\textit{deq}}(\,)
\end{aligned}$$

Null actions, represented as $\epsilon$, on a state element are omitted from the action lists above. (The full action list for the *Add Execute* rule is $\langle\, \mathbf{a}_{\mathsf{pc}}, \mathbf{a}_{\mathsf{rf}}, \mathbf{a}_{\mathsf{bf}}, \mathbf{a}_{\mathsf{imem}}, \mathbf{a}_{\mathsf{dmem}} \,\rangle$ where $\mathbf{a}_{\mathsf{pc}}$, $\mathbf{a}_{\mathsf{imem}}$ and $\mathbf{a}_{\mathsf{dmem}}$ are $\epsilon$'s.) Also notice, each operation's $\pi$ expression has been augmented with an explicit test, ***notfull***(bf) or ***notempty***(bf), depending on how the operation accesses bf.

## 2.4 Another Pipelined Processor

This example demonstrates the versatility of operation-centric descriptions by deriving a variation of the pipelined processor. In the previous example, instruction decode and register file read are performed in the execute stage. The processor described in this example performs instruction decode and register read in the fetch stage instead.

The new two-stage pipelined processor is specified as a TRS whose terms have the signature Proc(pc, rf, bf, imem, dmem). Decoded instructions and their operand values are stored as instruction templates in the pipeline buffer bf. The instruction template terms have the signatures: TLoadi(rd,v), TAdd(rd,v1,v2), TSub(rd,v1,v2), TBz(vc,va), TLoad(rd,va), and TStore(va,v).

Below, the *Add* rule from Section 2.2 is split into two sub-rules such that instruction decode is included in the fetch-stage rule. The fetch-stage sub-rule is

*Add Fetch Rule:*
```
     Proc(pc, rf, bf, imem, dmem)
         if (r1∉Target(bf)) ∧ (r2∉Target(bf)) where Add(rd,r1,r2)=imem[pc]
→    Proc(pc+1, rf, bf;TAdd(rd,rf[r1],rf[r2]), imem, dmem)
```

The execute-stage sub-rule is

*Add Execute Rule:*
```
     Proc(pc, rf, itemp;bf, imem, dmem) where TAdd(rd,v1,v2)=itemp
→    Proc(pc, rf[rd:=(v1+v2)], bf, imem, dmem)
```

Splitting an operation into sub-operations destroys the atomicity of the original operation and can cause new behaviors that are not part of the original description. Thus, the sub-operations may need to resolve newly created hazards. In this case, the *Add Fetch* rule's predicate expression has been extended to check if the source register names, r1 and r2, are in *Target*(bf).[3] This extra condition stalls instruction fetching when a RAW (read-after-write) hazard exists.

As another example, consider the *Bz Taken* rule and the *Bz Not-Taken* rule from Section 2.2. Again, the rules can be split into their fetch and execute sub-operations. Both *Bz* rules share the following instruction fetch rule:

*Bz Fetch Rule:*
```
     Proc(pc, rf, bf, imem, dmem)
         if (rc∉Target(bf)) ∧ (ra∉Target(bf)) where Bz(rc,ra)=imem[pc]
→    Proc(pc+1, rf, bf;TBz(rf[rc],rf[ra]), imem, dmem)
```

The two execute rules for the *Bz* instruction template are

*Bz Taken Execute Rule:*
```
     Proc(pc, rf, itemp;bf, imem, dmem) if (vc=0) where TBz(vc,va)=itemp
→    Proc(va, rf, ϵ, imem, dmem)
```

and

*Bz Not-Taken Execute Rule:*
```
     Proc(pc, rf, itemp;bf, imem, dmem) if (vc≠0) where TBz(vc,va)=itemp
→    Proc(pc, rf, bf, imem, dmem)
```

As in the previous examples, the operations in this example can also be restated

[3] *Target*(bf) is a shorthand for the set of destination register names in bf. Let bf contain entries $i_1$;...;$i_n$. 'r∉*Target*(bf)' stands for '(r≠*Dest*($i_1$))∧...∧(r≠*Dest*($i_n$))' where *Dest*(itemp) extracts the destination register name of an instruction template.

in terms of actions:

$$
\begin{aligned}
\pi_{Add-Fetch} &= (\mathsf{imem[pc]{=}Add(rd,r1,r2)}) \\
&\qquad \wedge(\mathsf{r1}\notin \mathit{Target}(\mathsf{bf}))\wedge(\mathsf{r2}\notin \mathit{Target}(\mathsf{bf}))\wedge\boldsymbol{notfull}(\mathsf{bf}) \\
\mathbf{a}_{Add-Fetch,\mathsf{pc}} &= \boldsymbol{set}(\mathsf{pc{+}1}) \\
\mathbf{a}_{Add-Fetch,\mathsf{bf}} &= \boldsymbol{enq}(\mathsf{TAdd(rd,rf[r1],rf[r2])})
\end{aligned}
$$

$$
\begin{aligned}
\pi_{Add-Exec} &= (\boldsymbol{first}(\mathsf{bf}){=}\mathsf{TAdd(rd,v1,v2)})\wedge\boldsymbol{notempty}(\mathsf{bf}) \\
\mathbf{a}_{Add-Exec,\mathsf{rf}} &= \boldsymbol{a\text{-}set}(\mathsf{rd,v1{+}v2}) \\
\mathbf{a}_{Add-Exec,\mathsf{bf}} &= \boldsymbol{deq}(\,)
\end{aligned}
$$

$$
\begin{aligned}
\pi_{Bz-Fetch} &= (\mathsf{imem[pc]{=}Bz(rc,ra)}) \\
&\qquad \wedge(\mathsf{rc}\notin \mathit{Target}(\mathsf{bf}))\wedge(\mathsf{ra}\notin \mathit{Target}(\mathsf{bf}))\wedge\boldsymbol{notfull}(\mathsf{bf}) \\
\mathbf{a}_{Bz-Fetch,\mathsf{pc}} &= \boldsymbol{set}(\mathsf{pc{+}1}) \\
\mathbf{a}_{Bz-Fetch,\mathsf{bf}} &= \boldsymbol{enq}(\mathsf{TBz(rf[rc],rf[ra])})
\end{aligned}
$$

$$
\begin{aligned}
\pi_{BzTaken-Exec} &= (\boldsymbol{first}(\mathsf{bf}){=}\mathsf{TBz(vc,vt)})\wedge(\mathsf{vc{=}0})\wedge\boldsymbol{notempty}(\mathsf{bf}) \\
\mathbf{a}_{BzTaken-Exec,\mathsf{pc}} &= \boldsymbol{set}(\mathsf{va}) \\
\mathbf{a}_{BzTaken-Exec,\mathsf{bf}} &= \boldsymbol{clear}(\,)
\end{aligned}
$$

$$
\begin{aligned}
\pi_{BzNotTaken-Exec} &= (\boldsymbol{first}(\mathsf{bf}){=}\mathsf{TBz(vc,vt)})\wedge(\mathsf{vc{\neq}0})\wedge\boldsymbol{notempty}(\mathsf{bf}) \\
\mathbf{a}_{BzNotTaken-Exec,\mathsf{bf}} &= \boldsymbol{deq}(\,)
\end{aligned}
$$

# Chapter 3

# Hardware Synthesis and Scheduling

Implementing an operation-centric description involves combining the atomic operations into a single coherent next-state logic for a state machine. For performance reasons, an implementation should carry out multiple operations concurrently while still maintaining a behavior that is consistent with a sequential execution of the atomic operations. This chapter first presents the Abstract Transition System (ATS), an abstract state machine whose transitions are specified operation-centrically. The chapter next develops the procedure to synthesize an ATS into an efficient synchronous digital implementation. A straightforward implementation that only executes one operation per clock cycle is presented first. This is followed by two increasingly optimized implementations that support the concurrent execution of operations.

## 3.1 Abstract Transition Systems

ATS is designed to be the intermediate target in the compilation of source-level operation-centric hardware description languages. An ATS is defined by $\mathcal{S}$, $\mathcal{S}^o$ and $\mathcal{X}$. $\mathcal{S}$ is a list of state elements, and $\mathcal{S}^o$ is a list of initial values for the elements in $\mathcal{S}$. $\mathcal{X}$ is a list of operation-centric transitions. The structure of an ATS is summarized in Figure 3-1.

### 3.1.1 State Elements

This thesis defines a restricted ATS where an element in $\mathcal{S}$ can only be a register, an array, or a FIFO. For generality, the ATS abstraction can be extended with new elements and variations on existing elements.

A register $\mathsf{R}$ can store an integer value up to a specified maximum word size. The value stored in $\mathsf{R}$ can be referenced in an expression using the side-effect-free $\mathsf{R}.\boldsymbol{get}(\,)$ query operator. For conciseness, $\mathsf{R}.\boldsymbol{get}(\,)$ can be abbreviated simply as $\mathsf{R}$ in an expression. $\mathsf{R}$'s content can be set to *value* by the $\mathsf{R}.\boldsymbol{set}(\mathit{value})$ action operator. Any number of queries are allowed in an atomic update step, but each register allows

$$
\begin{array}{rcl}
\mathcal{ATS} & = & \langle\, \mathcal{S},\, \mathcal{S}^o,\, \mathcal{X} \,\rangle \\
\mathcal{S} & = & \langle\, \mathsf{R}_1,...,\mathsf{R}_{NR},\mathsf{A}_1,...,\mathsf{A}_{NA},\mathsf{F}_1,...,\mathsf{F}_{NF},\mathsf{O}_1,...,\mathsf{O}_{NO},\mathsf{I}_1,...,\mathsf{I}_{NI} \,\rangle \\
\mathcal{S}^o & = & \langle\, v^{R_1},...,v^{R_{NR}},v^{A_1},...,v^{A_{NA}},v^{F_1},...,v^{F_{NF}},v^{O_1},...,v^{O_{NO}} \,\rangle \\
\mathcal{X} & = & \{\, T_1,...,T_M \,\} \\
T & = & \langle\, \pi,\, \alpha \,\rangle \\
\pi & = & \mathit{exp} \\
\alpha & = & \langle\, \mathbf{a}^{R_1},...,\mathbf{a}^{R_{NR}},\mathbf{a}^{A_1},...,\mathbf{a}^{A_{NA}},\mathbf{a}^{F_1},...,\mathbf{a}^{F_{NF}},\mathbf{a}^{O_1},...,\mathbf{a}^{O_{NO}},\mathbf{a}^{I_1},...,\mathbf{a}^{I_{NI}} \,\rangle \\
\mathbf{a}^R & = & \epsilon \;[]\; \boldsymbol{set}(\mathit{exp}) \\
\mathbf{a}^A & = & \epsilon \;[]\; \boldsymbol{a\text{-}set}(\mathit{exp}_{idx},\, \mathit{exp}_{data}) \\
\mathbf{a}^F & = & \epsilon \;[]\; \boldsymbol{enq}(\mathit{exp}) \;[]\; \boldsymbol{deq}(\,) \;[]\; \boldsymbol{en\text{-}deq}(\mathit{exp}) \;[]\; \boldsymbol{clear}(\,) \\
\mathbf{a}^O & = & \epsilon \;[]\; \boldsymbol{set}(\mathit{exp}) \\
\mathbf{a}^I & = & \epsilon \\
\mathit{exp} & = & \mathit{constant} \;[]\; \mathit{Primitive\text{-}Op}(\mathit{exp}_1,\, ...,\, \mathit{exp}_n) \\
 & [] & \mathsf{R}.\boldsymbol{get}(\,) \;[]\; \mathsf{A}.\boldsymbol{a\text{-}get}(\mathit{idx}) \\
 & [] & \mathsf{F}.\boldsymbol{first}(\,) \;[]\; \mathsf{F}.\boldsymbol{notfull}(\,) \;[]\; \mathsf{F}.\boldsymbol{notempty}(\,) \\
 & [] & \mathsf{O}.\boldsymbol{get}(\,) \;[]\; \mathsf{I}.\boldsymbol{get}(\,)
\end{array}
$$

Figure 3-1: ATS summary.

at most one ***set*** action in each atomic update step. An atomic update step is defined in Section 3.1.3 where the operational semantics of an ATS is defined.

An array A can store a specified number of values. The value stored in the *idx*'th entry of A can be referenced in an expression using the side-effect-free A.***a-get***(*idx*) query operator. A.***a-get***(*idx*) can be abbreviated as A[*idx*]. The content of the *idx*'th entry of A can be set to *value* using the A.***a-set***(*idx*,*value*) action operator. Out-of-bound queries or actions on an array are not allowed. In this thesis, each array allows at most one ***a-set*** action in each atomic update step, but any number of queries on an array are allowed. In general, variations of the array elements can be defined to allow multiple ***a-set*** actions or to limit the number of ***a-get*** queries.

A FIFO F stores a finite but unspecified number of values in a first-in, first-out manner. The oldest value in F can be referenced in an expression using the side-effect-free F.***first***( ) query operator, and can be removed by the F.***deq***( ) action operator. A new *value* can be added to F using the F.***enq***(*value*) action operator. F.***en-deq***(*value*) is a compound action that enqueues a new value after dequeuing the oldest value. In addition, the entire contents of F can be cleared using the F.***clear***( ) action operator. Underflowing or overflowing a FIFO by an ***enq*** or a ***deq*** action is not allowed. The status of F can be queried using the side-effect-free F.***notfull***( ) and F.***notempty***( ) query operators that return a Boolean status. Each FIFO allows at most one action in each atomic update step, but any number of queries are allowed. Again, variations of FIFO elements can be defined to support multiple actions per update step.

Input elements and output elements are register-like state elements. An input element I is like a register without the ***set*** operator. I.***get***( ) returns the value of an external input port instead of the content of an actual register. An output element O supports both ***set*** and ***get***, and its content is visible outside of the ATS on an output port. Input and output elements can be treated exactly like registers except when the input and output ports are instantiated in the final phase of circuit synthesis (described in Section 3.2.3).

### 3.1.2 State Transitions

An element in $\mathcal{X}$ is a transition. A transition is a pair, $\langle \pi, \alpha \rangle$. $\pi$ is a Boolean value expression. In general, a value expression can contain arithmetic and logical operations on scalar values. A value in an expression can be a constant or a value queried from a state element. Given an ATS whose $\mathcal{S}$ has $N$ elements, $\alpha$ is a list of $N$ actions, one for each state element. An action is specified as an action operator plus its arguments in terms of value expressions. A null action is represented by the symbol '$\epsilon$'. For all actions in $\alpha$, the $i$'th action of $\alpha$ must be valid for the $i$'th state element in $\mathcal{S}$.

### 3.1.3 Operational Semantics

The initial value of the $i$'th state element (except for input elements) is taken from the $i$'th element of $\mathcal{S}^o$. From this initial state, the execution of an ATS takes place as a sequence of state transitions in atomic update steps.

At the start of an atomic update step, all $\pi$ expressions in $\mathcal{X}$ are evaluated using the current contents of the state elements. In a given step, an applicable transition is one whose $\pi$ expression evaluates to true. All argument expressions to the actions in $\alpha$'s are also evaluated using the current state of $\mathcal{S}$.

At the end of an atomic update step, one transition is selected nondeterministically from the applicable transitions. $\mathcal{S}$ is then modified according to the actions of the selected transition. For each action in the selected $\alpha$, the $i$'th action is applied to the $i$'th state element in $\mathcal{S}$, using the argument values evaluated at the beginning of the step. If an illegal action or combination of actions is performed on a state element, the system halts with an error. A valid ATS specification never produces an error.

The atomic update step repeats until $\mathcal{S}$ is in a state where none of the transitions are applicable. In this case, the system halts without an error. Alternatively, a sequence of transitions can lead $\mathcal{S}$ to a state where selecting any applicable transition leaves $\mathcal{S}$ unchanged. The system also halts without an error in this case. Some systems may never halt. Due to nondeterminism, some systems could halt in different states from one execution to the next.

### 3.1.4 Functional Interpretation

ATS is defined in terms of state elements and actions with side-effects. In some circumstances, it is convenient to have a functional interpretation of ATS where an execution produces a sequence of $\mathcal{S}$ values. In the functional domain, the state of $\mathcal{S}$ is represented by a collection of values. $\mathsf{R}$ is represented by an integer value, $\mathsf{A}$ is an array of values, and $\mathsf{F}$ is an ordered list of values. The effect of a transition on state

$\mathsf{s}$ can be expressed as a function of $\pi$ and $\delta$ such that

$$\lambda\ \mathsf{s}.\ \textit{if}\ \pi(\mathsf{s})\ \textit{then}\ \delta(\mathsf{s})\ \textit{else}\ \mathsf{s}$$

where $\delta$ is a function that computes a new $\mathcal{S}$ value based on the current $\mathcal{S}$ value. Given a transition whose $\alpha$ is $\langle\, \mathbf{a}^{R_1}...,\ \mathbf{a}^{A_1}...,\ \mathbf{a}^{F_1}...\,\rangle$, its $\delta$ in the functional domain is

```
δ(s)= let
          $\langle\, v^{R_1}...,\ v^{A_1}...,\ v^{F_1}...\,\rangle = \mathsf{s}$
      in
          $\langle\, \textsc{Apply}(\mathbf{a}^{R_1}, v^{R_1}),\ ...,\ \textsc{Apply}(\mathbf{a}^{A_1}, v^{A_1}),\ ...,\ \textsc{Apply}(\mathbf{a}^{F_1}, v^{F_1}),\ ...\,\rangle$
```

Applying an action in the functional domain entails computing the new value of a state element based on the old values of the state elements. Given a state element whose current value is $\mathsf{v}$, the state's new value after receiving action $\mathbf{a}$ can be computed by

```
$\textsc{Apply}(\mathbf{a}, \mathsf{v})$= case a of
        $\textbf{\textit{set}}(\mathsf{exp}) \Rightarrow \textsc{Eval}(\mathsf{exp})$
        $\textbf{\textit{a-set}}(\mathsf{exp}_{idx}, \mathsf{exp}_{data}) \Rightarrow \mathsf{v}[\textsc{Eval}(\mathsf{exp}_{idx}) := \textsc{Eval}(\mathsf{exp}_{data})]$
        $\textbf{\textit{enq}}(\mathsf{exp}) \Rightarrow \mathsf{v};\textsc{Eval}(\mathsf{exp})$
        $\textbf{\textit{deq}}(\,) \Rightarrow \textsc{RestOf}(\mathsf{v})$
        $\textbf{\textit{en-deq}}(\mathsf{exp}) \Rightarrow \textsc{RestOf}(\mathsf{v});\textsc{Eval}(\mathsf{exp})$
        $\textbf{\textit{clear}}(\,) \Rightarrow \textit{empty list}$
        $\epsilon \Rightarrow \mathsf{v}$
```

where $\textsc{RestOf}(\mathsf{list})$ is the remainder of $\mathsf{list}$ after the first element is removed.

**Example 3.1 (GCD)**

This ATS corresponds to the TRS described in Section 2.1. The ATS describes the computation of the greatest common divisor of two 32-bit integers using Euclid's Algorithm. $\mathcal{S}$ is $\langle\, \mathsf{R}_a,\ \mathsf{R}_b\,\rangle$ where $\mathsf{R}_a$ and $\mathsf{R}_b$ are 32-bit registers. $\mathcal{X}$ consists of two transition pairs, $\langle\, \pi_1,\ \alpha_1\,\rangle$ and $\langle\, \pi_2,\ \alpha_2\,\rangle$ where

$$\pi_1 = (\mathsf{R}_a \geq \mathsf{R}_b) \wedge (\mathsf{R}_b \neq 0)$$
$$\alpha_1 = \langle\, \textbf{\textit{set}}(\mathsf{R}_a\text{-}\mathsf{R}_b),\ \epsilon\,\rangle$$

$$\pi_2 = \mathsf{R}_a < \mathsf{R}_b$$
$$\alpha_2 = \langle\, \textbf{\textit{set}}(\mathsf{R}_b),\ \textbf{\textit{set}}(\mathsf{R}_a)\,\rangle$$

The two transitions correspond to the *Mod* operation and the *Flip* operation from Section 2.1, respectively. The initial values $\mathcal{S}^o = \langle\, 2,\ 4\,\rangle$ initialize $\mathsf{R}_a$ and $\mathsf{R}_b$ for the computation of Gcd(2, 4).

□

**Example 3.2 (A Simple Processor)**

This ATS corresponds to the ISA described in Section 2.2. The programmer-visible processor states are represented by $\mathcal{S}{=}\langle\, \mathsf{R}_{PC}, \mathsf{A}_{RF}, \mathsf{A}_{IMEM}, \mathsf{A}_{DMEM}\,\rangle$ where $\mathsf{R}_{PC}$ is a 16-bit program counter register, $\mathsf{A}_{RF}$ is a general purpose register file of four 16-bit values, $\mathsf{A}_{IMEM}$ is a $2^{16}$-entry array of 25-bit instruction words, and $\mathsf{A}_{DMEM}$ is a $2^{16}$-entry array of 16-bit values.

$\mathcal{X}$ consists of transitions that correspond to the rewrite rules from Section 2.2, i.e., each transition describes the execution of an instruction in the ISA. Let 0, 2 and 4 be the numerical values assigned to the instruction opcodes *Loadi* (Load Immediate), *Add* (Triadic Register Add) and *Bz* (Branch if Zero). Also, let the instruction word stored in $\mathsf{A}_{IMEM}$ be decoded as

$$\begin{aligned}
&\mathit{opcode} = \text{bits 24 down to 22 of } \mathsf{A}_{IMEM}[\mathsf{R}_{PC}]\\
&\mathit{rd} = \text{bits 21 down to 20 of } \mathsf{A}_{IMEM}[\mathsf{R}_{PC}]\\
&\mathit{r1} = \text{bits 19 down to 18 of } \mathsf{A}_{IMEM}[\mathsf{R}_{PC}]\\
&\mathit{r2} = \text{bits 17 down to 16 of } \mathsf{A}_{IMEM}[\mathsf{R}_{PC}]\\
&\mathit{const} = \text{bits 15 down to 0 of } \mathsf{A}_{IMEM}[\mathsf{R}_{PC}]
\end{aligned}$$

$\pi$'s and $\alpha$'s of the transitions corresponding to the execution of *Loadi*, *Add* and *Bz* are

$$\begin{aligned}
&\pi_{Loadi} = (\mathit{opcode}{=}0)\\
&\alpha_{Loadi} = \langle\, \textbf{\textit{set}}(\mathsf{R}_{PC}{+}1),\ \textbf{\textit{a-set}}(\mathit{rd},\ \mathit{const}),\ \epsilon,\ \epsilon\,\rangle\\[1em]
&\pi_{Add} = (\mathit{opcode}{=}2)\\
&\alpha_{Add} = \langle\, \textbf{\textit{set}}(\mathsf{R}_{PC}{+}1),\ \textbf{\textit{a-set}}(\mathit{rd},\ \mathsf{A}_{RF}[\mathit{r1}]{+}\mathsf{A}_{RF}[\mathit{r2}]),\ \epsilon,\ \epsilon\,\rangle\\[1em]
&\pi_{BzTaken} = (\mathit{opcode}{=}4) \wedge (\mathsf{A}_{RF}[\mathit{r1}]{=}0)\\
&\alpha_{BzTaken} = \langle\, \textbf{\textit{set}}(\mathsf{A}_{RF}[\mathit{r2}]),\ \epsilon,\ \epsilon,\ \epsilon\,\rangle\\[1em]
&\pi_{BzNotTaken} = (\mathit{opcode}{=}4) \wedge (\mathsf{A}_{RF}[\mathit{r1}]{\neq}0)\\
&\alpha_{BzNotTaken} = \langle\, \textbf{\textit{set}}(\mathsf{R}_{PC}{+}1),\ \epsilon,\ \epsilon,\ \epsilon\,\rangle
\end{aligned}$$

□

**Example 3.3 (A Pipelined Processor)**

This ATS corresponds to the two-stage pipelined processor from Section 2.3. $\mathcal{S}$ is $\langle\, \mathsf{R}_{PC}, \mathsf{A}_{RF}, \mathsf{F}_{BF}, \mathsf{A}_{IMEM}, \mathsf{A}_{DMEM}\,\rangle$. $\mathsf{F}_{BF}$ is the pipeline stage FIFO for holding fetched instruction words.

Separate transitions are used to describe the operations in the *Fetch* stage and the *Execute* stage of the pipeline. $\pi$ and $\alpha$ of the transition that correspond to instruction fetch are

$$\begin{aligned}
&\pi_{Fetch} = \mathsf{F}_{BF}.\textbf{\textit{notfull}}(\,)\\
&\alpha_{Fetch} = \langle\, \textbf{\textit{set}}(\mathsf{R}_{PC}{+}1),\ \epsilon,\ \textbf{\textit{enq}}(\mathsf{A}_{IMEM}[\mathsf{R}_{PC}]),\ \epsilon,\ \epsilon\,\rangle
\end{aligned}$$

Again, let 0, 2 and 4 be the numerical values assigned to the instruction opcodes *Loadi* (Load Immediate), *Add* (Triadic Register Add) and *Bz* (Branch if Zero), and let the instruction word returned by $\mathsf{F}_{BF}.\boldsymbol{first}(\,)$ be decoded as

$$\begin{aligned}
\mathit{opcode} &= \text{bits 24 down to 22 of } \mathsf{F}_{BF}.\boldsymbol{first}(\,)\\
\mathit{rd} &= \text{bits 21 down to 20 of } \mathsf{F}_{BF}.\boldsymbol{first}(\,)\\
\mathit{r1} &= \text{bits 19 down to 18 of } \mathsf{F}_{BF}.\boldsymbol{first}(\,)\\
\mathit{r2} &= \text{bits 17 down to 16 of } \mathsf{F}_{BF}.\boldsymbol{first}(\,)\\
\mathit{const} &= \text{bits 15 down to 0 of } \mathsf{F}_{BF}.\boldsymbol{first}(\,)
\end{aligned}$$

$\pi$'s and $\alpha$'s of the transitions that correspond to the execution of *Loadi*, *Add* and *Bz* in the *Execute* stage are

$$\begin{aligned}
\pi_{Loadi-Execute} &= \mathsf{F}_{BF}.\boldsymbol{notempty}(\,) \wedge (\mathit{opcode}{=}0)\\
\alpha_{Loadi-Execute} &= \langle\, \epsilon,\ \boldsymbol{a\text{-}set}(\mathit{rd},\ \mathit{const}),\ \boldsymbol{deq}(\,),\ \epsilon,\ \epsilon\,\rangle
\end{aligned}$$

$$\begin{aligned}
\pi_{Add-Execute} &= \mathsf{F}_{BF}.\boldsymbol{notempty}(\,) \wedge (\mathit{opcode}{=}2)\\
\alpha_{Add-Execute} &= \langle\, \epsilon,\ \boldsymbol{a\text{-}set}(\mathit{rd},\ \mathsf{A}_{RF}[\mathit{r1}]{+}\mathsf{A}_{RF}[\mathit{r2}]),\ \boldsymbol{deq}(\,),\ \epsilon,\ \epsilon\,\rangle
\end{aligned}$$

$$\begin{aligned}
\pi_{BzTaken-Execute} &= \mathsf{F}_{BF}.\boldsymbol{notempty}(\,) \wedge (\mathit{opcode}{=}4) \wedge (\mathsf{A}_{RF}[\mathit{r1}]{=}0)\\
\alpha_{BzTaken-Execute} &= \langle\, \boldsymbol{set}(\mathsf{A}_{RF}[\mathit{r2}]),\ \epsilon,\ \boldsymbol{clear}(\,),\ \epsilon,\ \epsilon\,\rangle
\end{aligned}$$

$$\begin{aligned}
\pi_{BzNotTaken-Execute} &= \mathsf{F}_{BF}.\boldsymbol{notempty}(\,) \wedge (\mathit{opcode}{=}4) \wedge (\mathsf{A}_{RF}[\mathit{r1}]{\neq}0)\\
\alpha_{BzNotTaken-Execute} &= \langle\, \epsilon,\ \epsilon,\ \boldsymbol{deq}(\,),\ \epsilon,\ \epsilon\,\rangle
\end{aligned}$$

□

## 3.2 Reference Implementation of an ATS

One straightforward implementation of an ATS is a finite state machine (FSM) that performs one atomic update step per clock cycle. The elements of $\mathcal{S}$, instantiated as clock synchronous registers, arrays and FIFOs, are the state of the FSM. (The hardware manifestations of the ATS state elements are shown in Figure 3-2.) The atomic transitions in $\mathcal{X}$ are combined to form the next-state logic of the FSM.

### 3.2.1 State Storage

The register primitive is rising-edge-triggered and has a clock enable. A register can take on a new value on every clock edge, but the current value of the register can fan out to many locations. This hardware usage paradigm is directly reflected in the definition of the $\mathsf{R}$ element of an ATS where only one ***set*** action is allowed in an atomic update step, but any number of ***get*** queries are allowed. In general, the capability of the storage primitives should match their respective ATS counterparts. When the ATS state element models are extended or modified, their hardware representations need to be adjusted accordingly, and vice versa.

The array primitive supports combinational reads, that is, the read-data port (RD) continuously outputs the value stored at the entry selected by the read-address

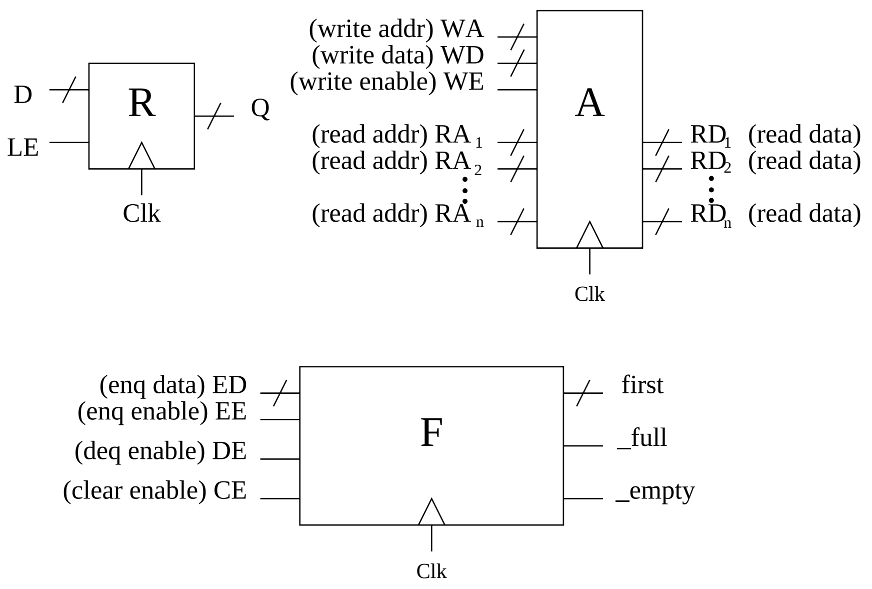


Figure 3-2: Synchronous state elements.

port (RA). The construction of the reference implementation assumes that an array primitive has as many read ports as necessary to support all ***a-get***( ) queries of ATS transitions. After a register-transfer level (RTL) design is generated, the actual number of combinational read ports can be reduced by common subexpression elimination. The array primitive only has one synchronous write port, allowing one update per clock cycle. In general, the number of array write ports should match the specified capability of the ATS array element A. The data at the write-data port (WD) is stored to the entry selected by the write-address port (WA) on a rising clock edge if the write-enable (WE) is asserted.

A FIFO primitive has three output ports that output the oldest value (first), a Boolean "not full" status (_full), and a Boolean "not empty" status (_empty). The FIFO primitive also has three synchronous interfaces that change the state of the FIFO by enqueuing a new value, dequeuing the oldest value, and clearing the entire FIFO. Each synchronous interface has an enable that must be asserted at the rising clock edge for the state change to take effect. New entries should not be enqueued to a FIFO if the FIFO is full. The exact size of the FIFO primitive does not have to be specified until the final design is simulated or synthesized.

### 3.2.2 State Transition Logic

The next-state logic can be derived from the transitions in $\mathcal{X}$ in three steps:

**Step 1:** All value expressions in the ATS are mapped to combinational logic signals on the current state of the state elements. In particular, this step creates a set of signals, $\pi_{T_1}$,...,$\pi_{T_M}$, that are the $\pi$ signals of transitions $T_1$,...,$T_M$ of an $M$-transition ATS. The logic mapping in this step assumes all required combinational resources are available. RTL optimizations can be employed to simplify the combinational logic

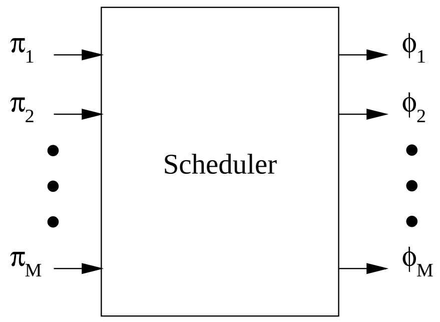


Figure 3-3: A monolithic scheduler for an $M$-transition ATS.

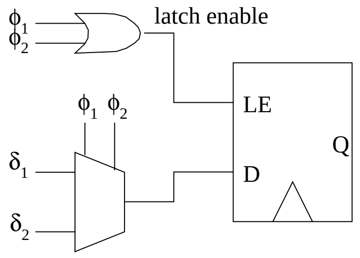


Figure 3-4: Circuits for merging two transitions' actions on the same register.

and to share duplicated logic.

**Step 2:** A scheduler is created to generate the set of arbitrated enable signals, $\phi_{T_1}$,...,$\phi_{T_M}$, based on $\pi_{T_1}$,...,$\pi_{T_M}$. (The block diagram of a scheduler is shown in Figure 3-3.) The reference implementation's scheduler asserts only one $\phi$ signal in each clock cycle, reflecting the selection of one applicable transition. A priority encoder is a valid scheduler for the reference implementation.

**Step 3:** One conceptually creates $M$ independent versions of next-state logic, each corresponding to one of the $M$ transitions in the ATS. Next, the $M$ sets of next-state logic are merged, state-element by state-element, using the $\phi$ signals for arbitration. For example, a register may have $N$ transitions that could affect it over time. ($N \leq M$ because some transitions may not affect the register.) The register takes on a new value if any of the $N$ relevant transitions is enabled in a clock cycle. Thus, the register's latch enable is the logical-*OR* of the $\phi$ signals of the $N$ relevant transitions. The new value of the register is selected from the $N$ candidate next-state values via a decoded multiplexer controlled by the $\phi$ signals. Figure 3-4 illustrates the merging circuit for a register that is affected by the ***set*** actions from two transitions. This merging scheme assumes at most one transition's action needs to be applied to a particular register element in a clock cycle. Furthermore, all actions of the same transition should be enabled everywhere in the same clock cycle to achieve the appearance of an atomic transition. The algorithm for generating the RTL description of a reference ATS implementation is given next.

### 3.2.3 RTL Description

**Scheduler:** Given an ATS with $M$ transitions $T_1$,...,$T_M$, a scheduler generates arbitrated transition enable signals $\phi_{T_1}$,...,$\phi_{T_M}$ where $\phi_{T_i}$ is used to select the actions of $T_i$. In any state $\mathsf{s}$, a valid scheduler must, at least, ensure that

1. $\phi_{T_i} \Rightarrow \pi_{T_i}(\mathsf{s})$
2. $\pi_{T_1}(\mathsf{s}) \vee \ldots \vee \pi_{T_M}(\mathsf{s}) \Rightarrow \phi_{T_1} \vee \ldots \vee \phi_{T_M}$

where $\pi_{T_i}(\mathsf{s})$ is the value of $T_i$'s $\pi$ expression in state $\mathsf{s}$. A priority encoder is a valid scheduler that selects one applicable transition per clock cycle. Since ATS allows nondeterminism in the selection, the priority encoder could use a static, round-robin or randomized priority.

**Register Update Logic:** Each $\mathsf{R}$ in $\mathcal{S}$ can be implemented using a synchronous register with clock enable (see Figure 3-2). For each $\mathsf{R}$ in $\mathcal{S}$, the set of transitions that update $\mathsf{R}$ is

$$\{\, T_{x_i} \mid \mathbf{a}^{R}_{T_{x_i}} = \boldsymbol{set}(exp_{x_i}) \,\}$$

where $\mathbf{a}^{R}_{T_{x_i}}$ is the action of $T_{x_i}$ on $\mathsf{R}$. The register's latch enable signal ($\mathsf{LE}$) is

$$\mathsf{LE} = \phi_{T_{x_1}} \vee \ldots \vee \phi_{T_{x_n}}$$

The register's data input signal ($\mathsf{D}$) is

$$\mathsf{D} = \phi_{T_{x_1}} \cdot exp_{x_1} + \ldots + \phi_{T_{x_n}} \cdot exp_{x_n}$$

The data input expression corresponds to a pass-gate multiplexer where $exp_{x_i}$ is enabled by $\phi_{T_{x_i}}$.

**Array Update Logic:** Each $\mathsf{A}$ in $\mathcal{S}$ can be implemented using a memory array with a synchronous write port (see Figure 3-2). (Given an array implementation with sufficient independent write ports, this scheme can also be generalized to support an ATS that allows multiple array writes in an atomic step.) For each $\mathsf{A}$ in $\mathcal{S}$, the set of transitions that write $\mathsf{A}$ is

$$\{\, T_{x_i} \mid \mathbf{a}^{A}_{T_{x_i}} = \boldsymbol{a\text{-}set}(idx_{x_i},\, data_{x_i}) \,\}$$

The array's write address ($\mathsf{WA}$) and data ($\mathsf{WD}$) are

$$\mathsf{WA} = \phi_{T_{x_1}} \cdot idx_{x_1} + \ldots + \phi_{T_{x_n}} \cdot idx_{x_n}$$

$$\mathsf{WD} = \phi_{T_{x_1}} \cdot data_{x_1} + \ldots + \phi_{T_{x_n}} \cdot data_{x_n}$$

The array's write enable signal ($\mathsf{WE}$) is

$$\mathsf{WE} = \phi_{T_{x_1}} \vee \ldots \vee \phi_{T_{x_n}}$$

**FIFO Update Logic:** Each $\mathsf{F}$ in $\mathcal{S}$ can be implemented using a first-in, first-out queue with synchronous enqueue, dequeue and clear interfaces (see Figure 3-2). For each $\mathsf{F}$, the inputs to the interfaces can be constructed as follows.

***Enqueue Interface:*** The set of transitions that enqueues a new value into $\mathsf{F}$ is

$$\{\, T_{x_i} \mid (\mathbf{a}^F_{T_{x_i}} = \boldsymbol{enq}(exp_{x_i})) \vee (\mathbf{a}^F_{T_{x_i}} = \boldsymbol{en\text{-}deq}(exp_{x_i})) \,\}$$

Every transition $T_{x_i}$ that enqueues into $\mathsf{F}$ is required to test $\mathsf{F}.\boldsymbol{notfull}(\,)$ in its $\pi$ expression. Hence, no $\phi_{T_{x_i}}$ will be asserted if $\mathsf{F}$ is already full. $\mathsf{F}$'s enqueue data ($\mathsf{ED}$) and enable ($\mathsf{EE}$) signals are

$$\mathsf{ED} = \phi_{T_{x_1}} \cdot exp_{x_1} + \ldots + \phi_{T_{x_n}} \cdot exp_{x_n}$$

$$\mathsf{EE} = \phi_{T_{x_1}} \vee \ldots \vee \phi_{T_{x_n}}$$

***Dequeue Interface:*** The set of transitions that dequeues from $\mathsf{F}$ is

$$\{\, T_{x_i} \mid (\mathbf{a}^F_{T_{x_i}} = \boldsymbol{deq}(\,)) \vee (\mathbf{a}^F_{T_{x_i}} = \boldsymbol{en\text{-}deq}(exp_{x_i})) \,\}$$

Every transition $T_{x_i}$ which dequeues from $\mathsf{F}$ is required to test $\mathsf{F}.\boldsymbol{notempty}(\,)$ in its $\pi$ expression. Similarly, no $\phi_{T_{x_i}}$ will be asserted if $\mathsf{F}$ is empty. $\mathsf{F}$'s dequeue enable ($\mathsf{DE}$) signal is

$$\mathsf{DE} = \phi_{T_{x_1}} \vee \ldots \vee \phi_{T_{x_n}}$$

***Clear Interface:*** The set of transitions that clears the contents of $\mathsf{F}$ is

$$\{\, T_{x_i} \mid \mathbf{a}^F_{T_{x_i}} = \boldsymbol{clear}(\,) \,\}$$

$\mathsf{F}$'s clear enable ($\mathsf{CE}$) is

$$\mathsf{CE} = \phi_{T_{x_1}} \vee \ldots \vee \phi_{T_{x_n}}$$

**Input and Output Ports:** Thus far, input/output elements, $\mathsf{I}$'s and $\mathsf{O}$'s, have been treated as $\mathsf{R}$'s. This is the only occasion where $\mathsf{I}$ and $\mathsf{O}$ elements require additional handling. To support output, the output of each $\mathsf{O}$ register is wired to an external output port of the same width. To support input, the net driven by the $\mathsf{Q}$ output of an $\mathsf{I}$ register is rewired to an external input port of the same width. The $\mathsf{I}$ register itself is only a placeholder structure that can be removed after its output connection has been rewired.

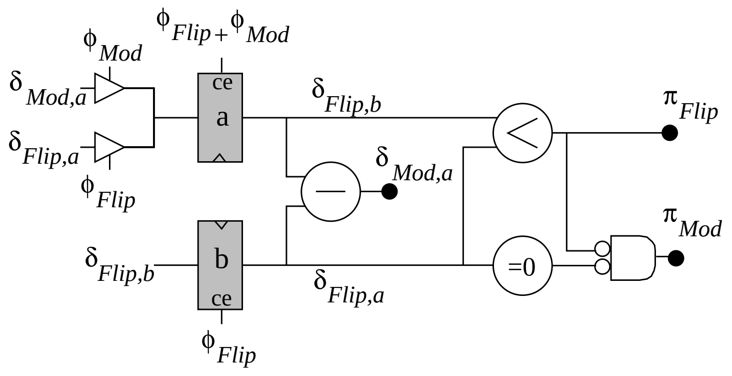


Figure 3-5: The GCD circuit from Example 3.4.

**Example 3.4 (GCD)**

A reference implementation of the ATS from Example 3.1 can be derived as follows.

**Scheduler:** The ATS consists of two transitions. For convenience, let the two transitions be named *Mod* and *Flip*. A two-way priority encoder suffices as a simple scheduler. The inputs to the priority encoder are

$$\pi_{Mod} = (\mathsf{R}_a \geq \mathsf{R}_b) \wedge (\mathsf{R}_b \neq 0)$$
$$\pi_{Flip} = \mathsf{R}_a < \mathsf{R}_b$$

The outputs of the priority enconder are $\phi_{Mod}$ and $\phi_{Flip}$.

**Register Update Logic:** The ATS consists of two registers $\mathsf{R}_a$ and $\mathsf{R}_b$. $\mathsf{R}_a$ is updated by both transitions whereas $\mathsf{R}_b$ is only updated by the *Flip* transition. Thus, the latch enables for the two registers are

$$\mathsf{LE}_{\mathsf{R}_a} = \phi_{Mod} \vee \phi_{Flip}$$

$$\mathsf{LE}_{\mathsf{R}_b} = \phi_{Flip}$$

The registers' data inputs are

$$\mathsf{D}_{\mathsf{R}_a} = \phi_{Mod} \cdot (\mathsf{R}_a\text{-}\mathsf{R}_b) + \phi_{Flip} \cdot (\mathsf{R}_b)$$

$$\mathsf{D}_{\mathsf{R}_b} = \mathsf{R}_a$$

$\mathsf{R}_b$ is only updated by one rule, and hence, multiplexing of its data input is not required. The synthesized circuit is depicted in Figure 3-5. (The scheduler is not shown.)

□

### 3.2.4 Correctness of a Synchronous Implementation

An implementation is said to implement an ATS correctly if

1. The implementation's sequence of state transitions corresponds to some execution of the ATS.
2. The implementation maintains liveness of state transitions.

Suppose for any $T$ the next-state logic of the reference implementation produces the same state changes as an application of $\alpha_T$, provided $\phi_T$ is the only $\phi$ signal asserted at the clock edge, the correctness of the reference implementation can be argued from the properties of the priority-encoder scheduler. First, the priority encoder maintains liveness because the encoder asserts a $\phi$ signal whenever at least one $\pi$ signal is asserted. (In general, liveness is maintained by any scheduler that satisfies the condition $\pi_{T_1}(s) \vee ... \vee \pi_{T_M}(s) \Rightarrow \phi_{T_1} \vee ... \vee \phi_{T_M}$.) Second, an implementation clock cycle corresponds exactly to an atomic update step of the ATS because the priority encoder selects only one of the applicable transitions to update the state at each clock edge. The sequence of state values constructed by sampling the reference implementation after every clock edge must correspond to some allowed sequence of ATS state transitions.

A dimension of correctness that is not addressed in the requirements above is the treatment of nondeterminism in an ATS. Unless the priority encoder in the reference implementation has true randomization, the reference implementation is deterministic, that is, the implementation can only embody one of the behaviors allowed by the ATS. Thus, an implementation can enter a livelock if the ATS depends on nondeterminism to make progress. The reference implementation cannot guarantee strong-fairness in the selection of transitions, that is, the reference implementation cannot guarantee any one transition will always be selected eventually, regardless of how many times the transition becomes applicable. However, using a round-robin priority encoder as the scheduler is sufficient to ensure weak-fairness, that is, if a transition remains applicable over a bounded number of consecutive steps, it will be selected at least once.

### 3.2.5 Performance Considerations

In a given atomic update step, if two simultaneously applicable transitions read and write mutually disjoint parts of $\mathcal{S}$, then the two transitions can be executed in any order in two successive steps to produce the same final state. In this scenario, although the ATS operational semantics prescribes a sequential execution in two atomic update steps, a synchronous hardware implementation can exploit the underlying parallelism and execute the two transitions concurrently in one clock cycle. In the case of the examples from Sections 2.3 and 2.4 where pipelined processors are described operation-centrically by stages, it is necessary to execute transitions from different pipeline stages concurrently to achieve pipelined executions. In general, it is dangerous to let two arbitrary transitions execute in the same clock cycle because of possible data dependence and structural conflicts. The next two sections, 3.3 and 3.4, formalize the conditions for the concurrent execution of transitions and suggest implementations with more aggressive scheduling schemes that execute multiple transitions in the same clock cycle.

## 3.3 Optimization I: Parallel Compositions

In a multiple-transitions-per-cycle implementation, the state transition in each clock cycle must correspond to a sequential execution of the ATS transitions in some order. If two transitions $T_a$ and $T_b$ become applicable in the same clock cycle when $\mathcal{S}$ is in state $s$, $\pi_{T_a}(\delta_{T_b}(s))$ or $\pi_{T_b}(\delta_{T_a}(s))$ must be true for an implementation to correctly select both transitions for execution. Otherwise, executing both transitions would be inconsistent with any sequential execution in two atomic update steps.

Given $\pi_{T_a}(\delta_{T_b}(s))$ or $\pi_{T_b}(\delta_{T_a}(s))$, there are two approaches to execute the actions of $T_a$ and $T_b$ in the same clock cycle. The first approach cascades the combinational logic from the two transitions. However, arbitrary cascading does not always improve the overall performance since it may lead to a longer cycle time. In a more practical approach, $T_a$ and $T_b$ are executed in the same clock cycle only if the correct final state can be constructed from an independent evaluation of their combinational logic on the same starting state.

### 3.3.1 Conflict-Free Transitions

Based on the intuition above, this section develops a scheduling algorithm based on the *conflict-free* relationship ($<>_{CF}$). $<>_{CF}$ is a symmetrical relationship that imposes a stronger requirement than necessary for executing two transitions concurrently. However, the symmetry of $<>_{CF}$ permits a straightforward implementation that concurrently executes multiple transitions if they are pairwise conflict-free. (Section 3.4 will present an improvement based on a more exact condition.) The conflict-free relationship and the parallel composition function $\boldsymbol{PC}$ are defined in Definition 3.1 and Definition 3.2.

**Definition 3.1 (Conflict-Free Relationship)**

Two transitions $T_a$ and $T_b$ are said to be conflict-free ($T_a <>_{CF} T_b$) if

$$\begin{aligned}\forall\ s.\ \pi_{T_a}(s) \wedge \pi_{T_b}(s) \Rightarrow\ & \pi_{T_b}(\delta_{T_a}(s)) \wedge \pi_{T_a}(\delta_{T_b}(s)) \wedge \\ & (\delta_{T_b}(\delta_{T_a}(s)) = \delta_{T_a}(\delta_{T_b}(s)) = \delta_{PC}(s))\end{aligned}$$

where $\delta_{PC}$ is the functional equivalent of $\boldsymbol{PC}(\alpha_{T_a}, \alpha_{T_b})$.

$\square$

**Definition 3.2 (Parallel Composition)**

$\boldsymbol{PC}(\alpha_a,\ \alpha_b)=\langle\ \boldsymbol{pc}_R(\mathsf{a}^{R_1},\ \mathsf{b}^{R_1}),\ ...,\ \boldsymbol{pc}_A(\mathsf{a}^{A_1},\ \mathsf{b}^{A_1}),\ ...,\ \boldsymbol{pc}_F(\mathsf{a}^{F_1},\ \mathsf{b}^{F_1}),\ ...\ \rangle$

*where* $\alpha_a=\langle\ \mathsf{a}^{R_1},\ ...,\ \mathsf{a}^{A_1},\ ...,\ \mathsf{a}^{F_1},\ ...\ \rangle$, $\alpha_b=\langle\ \mathsf{b}^{R_1},\ ...,\ \mathsf{b}^{A_1},\ ...,\ \mathsf{b}^{F_1},\ ...\ \rangle$

```
pc_R(a, b)=case a, b of a, ε ⇒ a
                        ε, b ⇒ b
                        ... ⇒ undefined
pc_A(a, b)=case a, b of a, ε ⇒ a
                        ε, b ⇒ b
                        ... ⇒ undefined
pc_F(a, b)=case a, b of a, ε ⇒ a
                        ε, b ⇒ b
                        enq(exp), deq( ) ⇒ en-deq(exp)
                        deq( ), enq(exp) ⇒ en-deq(exp)
                        ... ⇒ undefined
```

□

The function $\boldsymbol{PC}$ computes a new $\alpha$ by composing two $\alpha$'s that do not contain conflicting actions on the same state element. It can be shown that $\boldsymbol{PC}$ is commutative and associative.

Suppose $T_a$ and $T_b$ become applicable in the same state $\mathsf{s}$. $T_a <>_{CF} T_b$ implies that the two transitions can be applied in either order in two successive steps to produce the same final state $\mathsf{s}'$. $T_a <>_{CF} T_b$ further implies that an implementation could produce $\mathsf{s}'$ by applying the parallel composition of $\alpha_{T_a}$ and $\alpha_{T_b}$ to the same initial state $\mathsf{s}$. Theorem 3.1 extends this result to multiple pairwise conflict-free transitions.

**Theorem 3.1 (Composition of $<>_{CF}$ Transitions)**

Given a collection of $n$ transitions that are applicable in state $\mathsf{s}$, if all $n$ transitions are pairwise conflict-free, then the following condition holds for any ordering of $T_{x_1}$,...,$T_{x_n}$.

$$\pi_{T_{x_2}}(\delta_{T_{x_1}}(\mathsf{s})) \wedge\ ...\ \wedge \pi_{T_{x_n}}(\delta_{T_{x_{n-1}}}(\ ...\ \delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(\mathsf{s})))\ ...\ )) \wedge$$
$$(\delta_{T_{x_n}}(\delta_{T_{x_{n-1}}}(\ ...\ \delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(\mathsf{s})))\ ...\ )) = \delta_{PC}(\mathsf{s}))$$

$\delta_{PC}$ is the functional equivalent of the parallel composition of $\alpha_{T_{x_1}}$,...,$\alpha_{T_{x_n}}$, in any order.

**Proof:** Suppose Theorem 3.1 holds for $n = K$.

**Part 1.** Given a collection of $K$+1 transitions, $T_1$,...,$T_{K-1}$,$T_K$,$T_{K+1}$, that are all applicable in state $\mathsf{s}$ and are pairwise conflict-free, it follows from the induction hypothesis that

$$\pi_{T_{x_K}}(\delta_{T_{x_{K-1}}}(\ ...\ \delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(\mathsf{s})))\ ...\ ))$$

and also

$$\pi_{T_{x_{K+1}}}(\delta_{T_{x_{K-1}}}(\ \dots\ \delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(s)))\ \dots\ ))$$

In other words, $\pi_{T_{x_K}}(s') \wedge \pi_{T_{x_{K+1}}}(s')$ where $s'$ is $\delta_{T_{x_{K-1}}}(\ \dots\ \delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(s)))\ \dots\ )$. It follows from the definition of $<>_{CF}$ that

$$\pi_{T_{x_{K+1}}}(\delta_{T_{x_K}}(s'))$$

and hence

$$\pi_{T_{x_{K+1}}}(\delta_{T_{x_K}}(\delta_{T_{x_{K-1}}}(\ \dots\ \delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(s)))\ \dots\ )))$$

**Part 2.** It also follows from the induction hypothesis that

$$\delta_{T_{x_K}}(\delta_{T_{x_{K-1}}}(\dots\delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(s)))\dots))=(\textsc{Apply}(\boldsymbol{PC}(\alpha_{T_{x_1}},\dots,\alpha_{T_{x_{K-1}}},\alpha_{T_{x_K}})))(s)$$

and

$$\delta_{T_{x_{K+1}}}(\delta_{T_{x_{K-1}}}(\dots\delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(s)))\dots))=(\textsc{Apply}(\boldsymbol{PC}(\alpha_{T_{x_1}},\dots,\alpha_{T_{x_{K-1}}},\alpha_{T_{x_{K+1}}})))(s)$$

Following the definition of $\boldsymbol{PC}(\,)$, one can conclude from the second statement above that 1. any state element $e$ acted on by $\alpha_{T_{x_{K+1}}}$ is not acted on by $\alpha_{T_{x_{K-1}}},\dots,\alpha_{T_{x_1}}$ and 2. the state of $e$ is the same after $\delta_{T_{x_{K+1}}}(s)$ as after $\delta_{T_{x_{K+1}}}(s')$ where $s'$ is $\delta_{T_{x_{K-1}}}(\ \dots\ \delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(s)))\ \dots\ )$.

It is already shown from the first part of the proof that $\pi_{T_{x_K}}(s') \wedge \pi_{T_{x_{K+1}}}(s')$. It follows from the definition of $<>_{CF}$ that $\delta_{T_{x_{K+1}}}(\delta_{T_{x_K}}(s')) = \delta_{T_{x_K}}(\delta_{T_{x_{K+1}}}(s'))$. Any state element $e$ acted on by $\alpha_{T_{x_{K+1}}}$ is not acted on by $\alpha_{T_{x_K}}$ and the state of $e$ is the same after $\delta_{T_{x_{K+1}}}(\delta_{T_{x_K}}(s'))$ as after $\delta_{T_{x_{K+1}}}(s')$, and thus the same as after $\delta_{x_{K+1}}(s)$. Hence,

$$\begin{aligned}
&\delta_{x_{K+1}}(\delta_{T_{x_K}}(\delta_{T_{x_{K-1}}}(\ \dots\ \delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(s)))\ \dots\ )))\\
=\ &\delta_{x_{K+1}}(\ (\textsc{Apply}(\boldsymbol{PC}(\alpha_{T_{x_1}},\ \dots,\ \alpha_{T_{x_{K-1}}},\ \alpha_{T_{x_K}})))(s))\\
=\ &(\textsc{Apply}(\boldsymbol{PC}(\alpha_{T_{x_1}},\ \dots,\ \alpha_{T_{x_{K-1}}},\ \alpha_{T_{x_K}},\alpha_{T_{x_{K+1}}}\ )))(s)
\end{aligned}$$

Therefore, the theorem holds for $n = K+1$ if the theorem holds for $n = K$. The theorem holds for $K = 2$ by the definition of $<>_{CF}$. By induction, Theorem 3.1 holds for any $n$ greater or equal to 2.

□

### 3.3.2 Static Deduction of $<>_{CF}$

The scheduling algorithm given later in this section can work with a conservative $<>_{CF}$ test, that is, if the conservative test fails to identify a pair of transitions as $<>_{CF}$, the algorithm might generate a less optimal but still correct implementation.

**Analysis based on Domain and Range:**

A static determination of $<>_{CF}$ can be made by comparing the domains and ranges of two transitions. The domain of a transition is the set of state elements in $\mathcal{S}$ "read" by the expressions in either $\pi$ or $\alpha$. The domain of a transition can be further subclassified as $\pi$-domain and $\alpha$-domain depending on whether the state element is read by the $\pi$-expression or an expression in $\alpha$.

The range of a transition is the set of state elements in $\mathcal{S}$ that are acted on by $\alpha$. For this analysis, the head and the tail of a FIFO are considered to be separate elements. The functions to extract the domain and range of a transition are defined below.

**Definition 3.3 (Domain of $\pi$ and $\alpha$)**

$$
\begin{array}{lll}
\boldsymbol{D}_e(\mathsf{exp}) = \textit{case } \mathsf{exp} \textit{ of} & & \\
 & \textit{constant} \Rightarrow \{\ \} & \\
 & \mathsf{R}.\boldsymbol{get}(\,) \Rightarrow \{\,\mathsf{R}\,\} & \\
 & \mathsf{A}.\boldsymbol{a\text{-}get}(\mathsf{idx}) \Rightarrow \{\,\mathsf{A}\,\} \cup \boldsymbol{D}_e(\mathsf{idx}) & \\
 & \mathsf{F}.\boldsymbol{first}(\,) \Rightarrow \{\,\mathsf{F}_{head}\,\} & \\
 & \mathsf{F}.\boldsymbol{notfull}(\,) \Rightarrow \{\,\mathsf{F}_{tail}\,\} & \\
 & \mathsf{F}.\boldsymbol{notempty}(\,) \Rightarrow \{\,\mathsf{F}_{head}\,\} & \\
 & Op(\mathsf{exp}_1, ..., \mathsf{exp}_n) \Rightarrow \boldsymbol{D}_e(\mathsf{exp}_1) \cup ... \cup \boldsymbol{D}_e(\mathsf{exp}_n) &
\end{array}
$$

$$
\begin{array}{lll}
\boldsymbol{D}_\alpha(\alpha) = \boldsymbol{D}_R(\mathbf{a}^{R_1}) \cup ... \cup \boldsymbol{D}_A(\mathbf{a}^{A_1}) \cup ... \cup \boldsymbol{D}_F(\mathbf{a}^{F_1}) \cup ... & & \\
 & \textit{where } \alpha = \langle\, \mathbf{a}^{R_1}, ..., \mathbf{a}^{A_1}, ..., \mathbf{a}^{F_1}, ... \rangle & \\
\boldsymbol{D}_R(\mathbf{a}) = \textit{case } \mathbf{a} \textit{ of } \epsilon \Rightarrow \{\ \} & & \\
 & \quad \boldsymbol{set}(\mathsf{exp}) \Rightarrow \boldsymbol{D}_e(\mathsf{exp}) & \\
\boldsymbol{D}_A(\mathbf{a}) = \textit{case } \mathbf{a} \textit{ of } \epsilon \Rightarrow \{\ \} & & \\
 & \quad \boldsymbol{a\text{-}set}(\mathsf{idx},\mathsf{data}) \Rightarrow \boldsymbol{D}_e(\mathsf{idx}) \cup \boldsymbol{D}_e(\mathsf{data}) & \\
\boldsymbol{D}_F(\mathbf{a}) = \textit{case } \mathbf{a} \textit{ of } \epsilon \Rightarrow \{\ \} & & \\
 & \quad \boldsymbol{enq}(\mathsf{exp}) \Rightarrow \boldsymbol{D}_e(\mathsf{exp}) & \\
 & \quad \boldsymbol{en\text{-}deq}(\mathsf{exp}) \Rightarrow \boldsymbol{D}_e(\mathsf{exp}) & \\
 & \quad \boldsymbol{deq}(\,) \Rightarrow \{\ \} & \\
 & \quad \boldsymbol{clear}(\,) \Rightarrow \{\ \} &
\end{array}
$$

□

**Definition 3.4 (Range of $\alpha$)**

$$
\begin{aligned}
&\boldsymbol{R}_\alpha(\alpha) = \boldsymbol{R}_R(\mathsf{a}^{R_1}) \cup \ldots \cup \boldsymbol{R}_A(\mathsf{a}^{A_1}) \cup \ldots \cup \boldsymbol{R}_F(\mathsf{a}^{F_1}) \cup \ldots \\
&\qquad\qquad \textit{where } \alpha{=}\langle\, \mathsf{a}^{R_1}, \ldots, \mathsf{a}^{A_1}, \ldots, \mathsf{a}^{F_1}, \ldots \rangle \\
&\boldsymbol{R}_R(\mathsf{a}^R) = \textit{case } \mathsf{a}^R \textit{ of } \epsilon \Rightarrow \{\ \} \\
&\qquad\qquad\qquad\qquad \boldsymbol{set}(\text{-}) \Rightarrow \{\,\mathsf{R}\,\} \\
&\boldsymbol{R}_A(\mathsf{a}^A) = \textit{case } \mathsf{a}^A \textit{ of } \epsilon \Rightarrow \{\ \} \\
&\qquad\qquad\qquad\qquad \boldsymbol{a\text{-}set}(\text{-}, \text{-}) \Rightarrow \{\,\mathsf{A}\,\} \\
&\boldsymbol{R}_F(\mathsf{a}^F) = \textit{case } \mathsf{a}^F \textit{ of } \epsilon \Rightarrow \{\ \} \\
&\qquad\qquad\qquad\qquad \boldsymbol{enq}(\text{-}) \Rightarrow \{\,\mathsf{F}_{tail}\,\} \\
&\qquad\qquad\qquad\qquad \boldsymbol{deq}(\,) \Rightarrow \{\,\mathsf{F}_{head}\,\} \\
&\qquad\qquad\qquad\qquad \boldsymbol{en\text{-}deq}(\text{-}) \Rightarrow \{\,\mathsf{F}_{head}, \mathsf{F}_{tail}\,\} \\
&\qquad\qquad\qquad\qquad \boldsymbol{clear}(\,) \Rightarrow \{\,\mathsf{F}_{head}, \mathsf{F}_{tail}\,\}
\end{aligned}
$$

□

Using $\boldsymbol{D}(\,)$ and $\boldsymbol{R}(\,)$, a sufficient condition that ensures two transitions are conflict-free is given in Theorem 3.2.

**Theorem 3.2 (Sufficient Condition for $<>_{CF}$)**

$$
\begin{aligned}
&(\ (\boldsymbol{D}(\pi_{T_a}) \cup \boldsymbol{D}(\alpha_{T_a}))\ \not\cap\ \boldsymbol{R}(\alpha_{T_b})\ )\ \wedge\ (\ (\boldsymbol{D}(\pi_{T_b}) \cup \boldsymbol{D}(\alpha_{T_b}))\ \not\cap\ \boldsymbol{R}(\alpha_{T_a})\ )\ \wedge \\
&(\ \boldsymbol{R}(\alpha_{T_a})\ \not\cap\ \boldsymbol{R}(\alpha_{T_b})\ ) \\
&\qquad \Rightarrow (T_a <>_{CF} T_b)
\end{aligned}
$$

□

If the domain and range of two transitions do not overlap, then the two transitions do not have any data dependence. Since their ranges do not overlap, a valid parallel composition of $\alpha_{T_a}$ and $\alpha_{T_b}$ must exist.

**Analysis based on Mutual Exclusion:**

If two transitions never become applicable on the same state, then they are said to be mutually exclusive.

**Definition 3.5 (Mutually Exclusive Relationship)**

$$T_a <>_{ME} T_b \textit{ if } \forall\ \mathsf{s}.\ \neg(\pi_{T_a}(\mathsf{s}) \wedge \pi_{T_b}(\mathsf{s}))$$

□

Two transitions that are $<>_{ME}$ satisfy the definition of $<>_{CF}$ trivially. An exact test for $<>_{ME}$ requires determining the satisfiability of the expression $(\pi_{T_a}(\mathsf{s}) \wedge \pi_{T_b}(\mathsf{s}))$. Fortunately, the $\pi$ expression is usually a conjunction of relational constraints on the current values of state elements. A conservative test that scans two $\pi$ expressions for contradicting constraints on any one state element works well in practice.

### 3.3.3 Scheduling of $<>_{CF}$ Transitions

Using Theorem 3.1, instead of selecting a single transition per clock cycle, a scheduler can select any number of applicable transitions that are pairwise conflict-free. In other words, in each clock cycle, the $\phi$'s should satisfy the condition:

$$\phi_{T_a} \wedge \phi_{T_b} \Rightarrow T_a <>_{CF} T_b$$

where $\phi_T$ is the arbitrated transition enable signal for transition $T$. Given a set of applicable transitions in a clock cycle, many different subsets of pairwise conflict-free transitions could exist. Selecting the optimum subset requires evaluating the relative importance of the transitions. Alternatively, an objective metric simply optimizes the number of transitions executed in each clock cycle.

**Partitioned Scheduler:**

In a partitioned scheduler, transitions in $\mathcal{X}$ are first partitioned into as many disjoint scheduling groups, $\mathcal{P}_1,...,\mathcal{P}_k$, as possible such that

$$(T_a \in \mathcal{P}_a) \wedge (T_b \in \mathcal{P}_b) \Rightarrow T_a <>_{CF} T_b$$

Transitions in different scheduling groups are conflict-free, and hence each scheduling group can be scheduled independently of other groups. For a given scheduling group containing $T_{x_1},...,T_{x_n}$, $\phi_{T_{x_1}},...,\phi_{T_{xn}}$ can be generated from $\pi_{T_{x_1}}(\mathsf{s}),...,\pi_{T_{xn}}(\mathsf{s})$ using a priority encoder. In the best case, a transition $T$ is conflict-free with every other transition in $\mathcal{X}$. In which case, $T$ is in a scheduling group by itself, and $\phi_T = \pi_T$ without arbitration.

$\mathcal{X}$ can be partitioned into scheduling groups by finding the connected components of an undirected graph whose nodes are transitions $T_1$, ..., $T_M$ and whose edges are $\{\,(T_i, T_j) \mid \neg(T_i <>_{CF} T_j)\,\}$. Each connected component is a scheduling group.

**Example 3.5 (Partitioned Scheduler)**

The undirected graph (a) in Figure 3-6 depicts the conflict-free relationships in an ATS with six transitions, $\mathcal{X}=\{\,T_1, T_2, T_3, T_4, T_5, T_6\,\}$. Each transition is represented as a node in the graph. Two transitions are conflict-free if their corresponding nodes are connected, i.e.,

$$\begin{array}{l}
(T_1 <>_{CF} T_2), (T_1 <>_{CF} T_3), (T_1 <>_{CF} T_5), (T_1 <>_{CF} T_6), \\
(T_2 <>_{CF} T_3), (T_2 <>_{CF} T_4), (T_2 <>_{CF} T_6), \\
(T_3 <>_{CF} T_4), (T_3 <>_{CF} T_5), (T_3 <>_{CF} T_6), \\
(T_4 <>_{CF} T_5), \\
(T_5 <>_{CF} T_6)
\end{array}$$

Graph (b) in Figure 3-6 gives the corresponding conflict graph where two nodes are connected if they are not conflict-free. The conflict graph has three connected

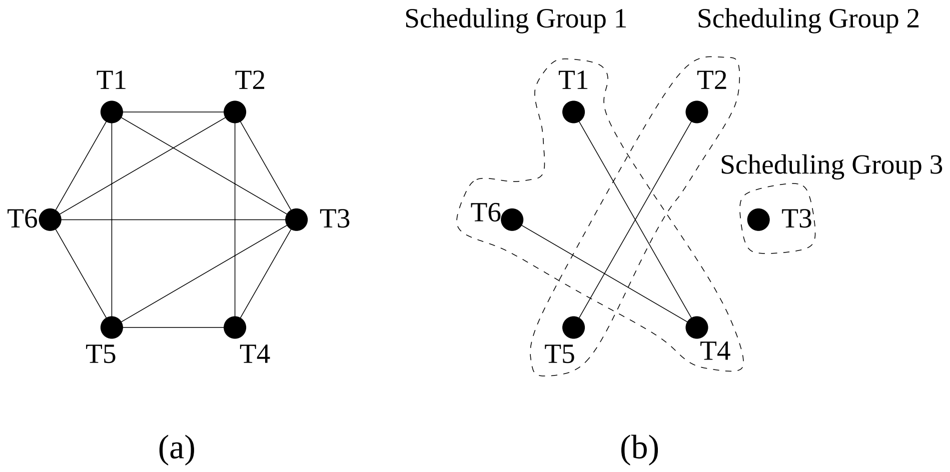


Figure 3-6: Scheduling conflict-free rules: (a) $<>_{CF}$ graph (b) Corresponding conflict graph and its connected components.

components, corresponding to the three $<>_{CF}$ scheduling groups. The $\phi$ signals corresponding to $T_1$, $T_4$ and $T_6$ can be generated using a priority encoding of their corresponding $\pi$'s. Scheduling group 2 also requires a scheduler to ensure $\phi_2$ and $\phi_5$ are not asserted in the same clock cycle. However, $\phi_{T_3}$=$\pi_{T_3}$ without any arbitration.

□

**Example 3.6 (Fully Partitioned Scheduling)**

In Example 3.4, a two-way priority scheduler is instantiated for a two-transition ATS. The two transitions' $\pi$-expressions are

$$\pi_{Mod} = (\mathsf{R}_a \geq \mathsf{R}_b) \wedge (\mathsf{R}_b \neq 0)$$
$$\pi_{Flip} = \mathsf{R}_a < \mathsf{R}_b$$

Notice, the two transitions' $\pi$-expressions have contradicting requirements on the value of $\mathsf{R}_a$. $\pi_{Flip}$ requires $\mathsf{R}_a \geq \mathsf{R}_b$, but $\pi_{Mod}$ requires $\mathsf{R}_a < \mathsf{R}_b$. Thus the two transitions are $<>_{ME}$ and therefore $<>_{CF}$. The transitions are each in its own scheduling group. Hence, $\phi_{Mod}$=$\pi_{Mod}$ and $\phi_{Flip}$=$\pi_{Flip}$ without any arbitration.

□

**Enumerated Scheduler:**

Scheduling group 1 of graph (b) in Figure 3-6 contains three transitions $\{T_1, T_4, T_6\}$ such that $T_1 <>_{CF} T_6$ but both $T_1$ and $T_6$ are not $<>_{CF}$ with $T_4$. Although the three transitions cannot be scheduled independently of each other, $T_1$ and $T_6$ can be selected together as long as $T_4$ is not selected in the same clock cycle. This selection is valid because $T_1$ and $T_6$ are $<>_{CF}$ between themselves and every transition selected by the other groups. In any given clock cycle, the scheduler for each group can independently select multiple transitions that are pairwise conflict-free within the scheduling group.

| $\pi_{T_1}$ | $\pi_{T_4}$ | $\pi_{T_6}$ | $\phi_{T_1}$ | $\phi_{T_4}$ | $\phi_{T_6}$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 |

Figure 3-7: Enumerated encoder table for Example 3.7.

For a scheduling group with transitions $T_{x_1}$,...,$T_{x_n}$, $\phi_{T_{x_1}}$,...,$\phi_{T_{x_n}}$ can be computed by a $2^n \times n$ lookup table indexed by $\pi_{T_{x_1}}(\mathsf{s})$,...,$\pi_{T_{x_n}}(\mathsf{s})$. The data value $d_1$,...,$d_n$ at the table entry with index $b_1$,...,$b_n$ can be determined by finding a clique in an undirected graph whose nodes $\mathcal{N}$ and edges $\mathcal{E}$ are defined as follows:

$$\begin{aligned}\mathcal{N} &= \{\, T_{x_i} \mid \text{index bit } b_i \text{ is asserted}\,\} \\ \mathcal{E} &= \{\, (T_{x_i}, T_{x_j}) \mid (T_{x_i} \in \mathcal{N}) \wedge (T_{x_j} \in \mathcal{N}) \wedge (T_{x_i} <>_{CF} T_{x_j})\,\}\end{aligned}$$

For each $T_{x_i}$ that is in the clique, assert data bit $d_i$.

**Example 3.7 (Enumerated Encoder)**

Scheduling group 1 from Example 3.5 can be scheduled by an enumerated encoder (Figure 3-7) that allows $T_1$ and $T_6$ to execute concurrently.

□

### 3.3.4 Performance Gain

When $\mathcal{X}$ can be partitioned into scheduling groups, the partitioned scheduler is smaller and faster than the monolithic encoder of the reference implementation from Section 3.2.3. The partitioned scheduler also reduces wiring cost and delay since $\pi$'s and $\phi$'s of unrelated transitions are not brought together for arbitration.

The property of the parallel composition function, $\boldsymbol{PC}$, ensures that transitions are conflict-free only if their actions on state elements do not conflict. For example, given a set of transitions that are all pairwise conflict-free, each $\mathsf{R}$ in $\mathcal{S}$ can be updated by at most one of those transitions at a time. Hence, the state update logic described in Section 3.2.3 can be used without any modification. Consequently, parallel composition of actions does not increase the combinational delay of the datapath. All in all, the implementation in this section achieves better performance than the reference implementation by allowing more transitions to execute in a clock cycle without increasing the cycle time.

## 3.4 Optimization II: Sequential Compositions

Consider the following example where $\boldsymbol{PC}(\alpha_{T_a},\alpha_{T_b})$ and its functional equivalent, $\delta_{PC}$, is well-defined for any choice of two transitions $T_a$ and $T_b$ from the following ATS:

$$\begin{array}{lcl} \mathcal{S} & = & \langle\, \mathsf{R}_1,\ \mathsf{R}_2,\ \mathsf{R}_3 \,\rangle \\ \mathcal{X} & = & \{\, \langle\, \mathit{true},\ \langle\, \boldsymbol{set}(\mathsf{R}_2{+}1),\ \epsilon,\ \epsilon \,\rangle\,\rangle \\ & & \ \ \langle\, \mathit{true},\ \langle\, \epsilon,\ \boldsymbol{set}(\mathsf{R}_3{+}1),\ \epsilon \,\rangle\,\rangle \\ & & \ \ \langle\, \mathit{true},\ \langle\, \epsilon,\ \epsilon,\ \boldsymbol{set}(\mathsf{R}_1{+}1) \,\rangle\,\rangle \,\} \end{array}$$

Although all transitions are always applicable, the $<>_{CF}$ scheduler proposed in the previous section would not permit $T_a$ and $T_b$ to execute in the same clock cycle because, in general, $\delta_{T_a}(\delta_{T_b}(\mathsf{s})) \neq \delta_{T_b}(\delta_{T_a}(\mathsf{s}))$. It can be shown that, for all $\mathsf{s}$, $\delta_{PC}(\mathsf{s})$ is consistent with at least one order of sequential execution of $T_a$ and $T_b$. Hence, their concurrent execution can be allowed in an implementation. However, the concurrent execution of all three transitions in a parallel composition does not always produce a consistent new state due to circular data dependencies among the three transitions. To capture these conditions, this section presents a more exact requirement for concurrent execution based on the sequential composibility relationship.

### 3.4.1 Sequentially-Composible Transitions

**Definition 3.6 (Sequential Composibility)**

Two transitions $T_a$ and $T_b$ are said to be sequentially composible ($T_a <_{SC} T_b$), if

$$\begin{array}{l} \forall\ \mathsf{s}.\ \pi_{T_a}(\mathsf{s}) \wedge \pi_{T_b}(\mathsf{s}) \Rightarrow \pi_{T_b}(\delta_{T_a}(\mathsf{s})) \wedge \\ \qquad\qquad\qquad\qquad\qquad (\delta_{T_b}(\delta_{T_a}(\mathsf{s})) = \delta_{SC}(\mathsf{s})) \end{array}$$

where $\delta_{SC}$ is the functional equivalent of $\boldsymbol{SC}(\alpha_{T_a},\ \alpha_{T_b})$.

□

**Definition 3.7 (Sequential Composition)**

$\boldsymbol{SC}(\alpha_a,\ \alpha_b)= \langle\, \boldsymbol{sc}_R(\mathbf{a}^{R_1},\ \mathbf{b}^{R_1}),\ ...,\ \boldsymbol{sc}_A(\mathbf{a}^{A_1},\ \mathbf{b}^{A_1}),\ ...,\ \boldsymbol{sc}_F(\mathbf{a}^{F_1},\ \mathbf{b}^{F_1}),\ ... \,\rangle$

*where* $\alpha_a=\langle\, \mathbf{a}^{R_1},\ ...,\ \mathbf{a}^{A_1},\ ...,\ \mathbf{a}^{F_1},\ ... \,\rangle$, $\alpha_b=\langle\, \mathbf{b}^{R_1},\ ...,\ \mathbf{b}^{A_1},\ ...,\ \mathbf{b}^{F_1},\ ... \,\rangle$

```
$\boldsymbol{sc}_R(\mathbf{a},\ \mathbf{b})=$ case $\mathbf{a},\ \mathbf{b}$ of
        $\mathbf{a},\ \epsilon \Rightarrow \mathbf{a}$
        $\epsilon,\ \mathbf{b} \Rightarrow \mathbf{b}$
        $... \Rightarrow \mathbf{b}$
```

```
$\boldsymbol{sc}_A(\mathsf{a}, \mathsf{b})$=case $\mathsf{a}, \mathsf{b}$ of
        $\mathsf{a}, \epsilon \Rightarrow \mathsf{a}$
        $\epsilon, \mathsf{b} \Rightarrow \mathsf{b}$
        $... \Rightarrow$ undefined

$\boldsymbol{sc}_F(\mathsf{a}, \mathsf{b})$=case $\mathsf{a}, \mathsf{b}$ of
        $\mathsf{a}, \epsilon \Rightarrow \mathsf{a}$
        $\epsilon, \mathsf{b} \Rightarrow \mathsf{b}$
        $\boldsymbol{enq}(\mathsf{exp}), \boldsymbol{deq}(\,) \Rightarrow \boldsymbol{en\text{-}deq}(\mathsf{exp})$
        $\boldsymbol{deq}(\,), \boldsymbol{enq}(\mathsf{exp}) \Rightarrow \boldsymbol{en\text{-}deq}(\mathsf{exp})$
        $\mathsf{a}, \boldsymbol{clear}(\,) \Rightarrow \boldsymbol{clear}(\,)$
        $... \Rightarrow$ undefined
```
□

The sequential composition function $\boldsymbol{SC}$ returns a new $\alpha$ by composing actions on the same element from two $\alpha$'s. $\boldsymbol{sc}_R$, $\boldsymbol{sc}_A$ and $\boldsymbol{sc}_F$ are the same as $\boldsymbol{pc}_R$, $\boldsymbol{pc}_A$ and $\boldsymbol{pc}_F$ except in two cases where $\boldsymbol{SC}$ allows two conflicting actions to be sequentialized. First, $\boldsymbol{sc}_R(\boldsymbol{set}(\mathsf{exp}_a), \boldsymbol{set}(\mathsf{exp}_b))$ is $\boldsymbol{set}(\mathsf{exp}_b)$ since the effect of the first action is overwritten by the second in a sequential application. Second, $\boldsymbol{sc}_F(\mathsf{a}, \boldsymbol{clear}(\,))$ returns $\boldsymbol{clear}(\,)$ since regardless of $\mathsf{a}$, applying $\boldsymbol{clear}(\,)$ leaves the FIFO emptied.

The $<_{SC}$ relationship is a relaxation of $<>_{CF}$. In particular, $<_{SC}$ is not symmetric. Suppose $T_a$ and $T_b$ are applicable in state $\mathsf{s}$. $T_a <_{SC} T_b$ only requires the concurrent execution of $T_a$ and $T_b$ on $\mathsf{s}$ to correspond to $\delta_{T_b}(\delta_{T_a}(\mathsf{s}))$, but not $\delta_{T_a}(\delta_{T_b}(\mathsf{s}))$. Two $<_{SC}$ transitions can also have conflicting actions that can be sequentialized. Theorem 3.3 extends this result to multiple transitions whose transitive closure on $<_{SC}$ is ordered.

**Theorem 3.3 (Composition of $<_{SC}$ Transitions)**

Given a sequence of $n$ transitions, $T_{x_1}$,...,$T_{x_n}$, that are all applicable in state $\mathsf{s}$, if $T_{x_j} <_{SC} T_{x_k}$ for all $j < k$ then

$$\begin{aligned}&\pi_{T_{x_2}}(\delta_{T_{x_1}}(\mathsf{s})) \wedge ... \wedge\\ &\pi_{T_{x_n}}(\delta_{T_{x_{n-1}}}( ... \delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(\mathsf{s}))) ... )) \wedge\\ &(\delta_{T_{x_n}}(\delta_{T_{x_{n-1}}}( ... \delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(\mathsf{s}))) ... )) = \delta_{SC}(\mathsf{s}))\end{aligned}$$

where $\delta_{SC}$ is the functional equivalent of the nested sequential composition $\boldsymbol{SC}( ... \boldsymbol{SC}(\boldsymbol{SC}(\alpha_{T_{x_1}}, \alpha_{T_{x_2}}), \alpha_{T_{x_3}}), ... )$.

**Proof:** The induction proof is similar to the proof of Theorem 3.1. Suppose Theorem 3.3 holds for $n = K$.

**Part 1.** Given a sequence of $K$+1 transitions, $T_{x_1}$,...,$T_{x_{K-1}}$,$T_{x_K}$,$T_{x_{K+1}}$, that are all applicable in state $\mathsf{s}$ and $T_{x_i} <_{SC} T_{x_j}$ for all $i < j$, it follows from the induction hypothesis that

$$\pi_{T_{x_K}}(\delta_{T_{x_{K-1}}}( ... \delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(\mathsf{s}))) ... ))$$

and also

$$\pi_{T_{x_{K+1}}}(\delta_{T_{x_{K-1}}}(\ ...\ \delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(\mathsf{s})))\ ...\ ))$$

In other words, $\pi_{T_{x_K}}(\mathsf{s}') \wedge \pi_{T_{x_{K+1}}}(\mathsf{s}')$ where $\mathsf{s}'$ is $\delta_{T_{x_{K-1}}}(\ ...\ \delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(\mathsf{s})))\ ...\ )$. It follows from the definition of $<_{SC}$ that

$$\pi_{T_{x_{K+1}}}(\delta_{T_{x_K}}(\mathsf{s}'))$$

and hence

$$\pi_{T_{x_{K+1}}}(\delta_{T_{x_K}}(\delta_{T_{x_{K-1}}}(\ ...\ \delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(\mathsf{s})))\ ...\ )))$$

**Part 2.** It also follows from the induction hypothesis that

$$\delta_{T_{x_K}}(\delta_{T_{x_{K-1}}}(...\delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(\mathsf{s})))...))=(\textsc{Apply}(\boldsymbol{SC}(\alpha_{T_{x_1}},...,\alpha_{T_{x_{K-1}}},\alpha_{T_{x_K}})))(\mathsf{s})$$

and

$$\delta_{T_{x_{K+1}}}(\delta_{T_{x_{K-1}}}(...\delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(\mathsf{s})))...))=(\textsc{Apply}(\boldsymbol{SC}(\alpha_{T_{x_1}},...,\alpha_{T_{x_{K-1}}},\alpha_{T_{x_{K+1}}})))(\mathsf{s})$$

Following the definition of $\boldsymbol{SC}(\,)$, one can conclude from the second statement above that 1. any state element $\mathsf{e}$ acted on by $\alpha_{T_{x_{K+1}}}$ can ignore the actions by $\alpha_{T_{x_{K-1}}},...,\alpha_{T_{x_1}}$ and 2. the state of $\mathsf{e}$ is the same after $\delta_{T_{x_{K+1}}}(\mathsf{s})$ as after $\delta_{T_{x_{K+1}}}(\mathsf{s}')$ where $\mathsf{s}'$ is $\delta_{T_{x_{K-1}}}(\ ...\ \delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(\mathsf{s})))\ ...\ )$.

It is already shown from the first part of the proof that $\pi_{T_{x_K}}(\mathsf{s}') \wedge \pi_{T_{x_{K+1}}}(\mathsf{s}')$. It follows from the definition of $<_{SC}$ that $\delta_{T_{x_{K+1}}}(\delta_{T_{x_K}}(\mathsf{s}')) = (\textsc{Apply}(\alpha_{T_{x_K}},\ \alpha_{T_{x_{K+1}}}))(\mathsf{s}')$. Any state element $\mathsf{e}$ acted on by $\alpha_{T_{x_{K+1}}}$ can ignore the actions by $\alpha_{T_{x_K}}$ and the state of $\mathsf{e}$ is the same after $\delta_{T_{x_{K+1}}}(\delta_{T_{x_K}}(\mathsf{s}'))$ as after $\delta_{T_{x_{K+1}}}(\mathsf{s}')$, and thus the same as after $\delta_{x_{K+1}}(\mathsf{s})$. Hence,

$$\begin{aligned} &\delta_{x_{K+1}}(\delta_{T_{x_K}}(\delta_{T_{x_{K-1}}}(\ ...\ \delta_{T_{x_3}}(\delta_{T_{x_2}}(\delta_{T_{x_1}}(\mathsf{s})))\ ...\ ))) \\ =\ &\delta_{x_{K+1}}(\ (\textsc{Apply}(\boldsymbol{SC}(\alpha_{T_{x_1}},\ ...,\ \alpha_{T_{x_{K-1}}},\ \alpha_{T_{x_K}})))(\mathsf{s})) \\ =\ &(\textsc{Apply}(\boldsymbol{SC}(\alpha_{T_{x_1}},\ ...,\ \alpha_{T_{x_{K-1}}},\ \alpha_{T_{x_K}},\ \alpha_{T_{x_{K+1}}}\ )))(\mathsf{s}) \end{aligned}$$

Therefore, the theorem holds for $n = K{+}1$ if the theorem holds for $n = K$. The theorem holds for $K = 2$ by the definition of $<_{SC}$. By induction, Theorem 3.3 holds for any $n$ greater or equal to 2.

□

### 3.4.2 Static Deduction of $<_{SC}$

Using $\boldsymbol{D}(\,)$ and $\boldsymbol{R}(\,)$, a sufficient condition for two transitions to be sequentially composible is given in Theorem 3.4.

**Theorem 3.4 (Sufficient Condition for $<_{SC}$)**

$$((\boldsymbol{D}(\pi_{T_b}) \cup \boldsymbol{D}(\alpha_{T_b})) \not\cap \boldsymbol{R}(\alpha_{T_a})) \wedge (\boldsymbol{SC}(\alpha_{T_a},\, \alpha_{T_b}) \textit{ is defined})$$
$$\Rightarrow T_a <_{SC} T_b$$

□

### 3.4.3 Scheduling of $<_{SC}$ Transitions

Incorporating the results from Theorem 3.3 into the partitioned scheduler from Section 3.3.3 allows additional transitions to execute in the same clock cycle. For each conflict-free scheduling group containing $T_{x_1}$,...,$T_{x_n}$, its scheduler generates the arbitrated transition enable signals $\phi_{T_{x_1}}$,...,$\phi_{T_{x_n}}$. In every $\mathsf{s}$, there must be an ordering of all asserted $\phi$'s, $\phi_{T_{y_1}}$,...,$\phi_{T_{y_m}}$, such that $T_{y_j} <_{SC} T_{y_k}$ if $j < k$. However, to simplify the state update logic, our algorithm further requires a statically chosen *SC-ordering* $T_{z_1}$,...,$T_{z_n}$ for each scheduling group such that

$$\forall\, \mathsf{s}.\ \phi_{T_{z_j}} \wedge \phi_{T_{z_k}} \Rightarrow T_{z_j} <_{SC} T_{z_k} \text{ if } j < k$$

**Scheduler:**

To construct the $<_{SC}$ scheduler for a conflict-free scheduling group that contains $T_{x_1}$,...,$T_{x_n}$, one first computes the group's SC-ordering using a topological sort on a directed graph whose nodes are $T_{x_1}$,...,$T_{x_n}$ and whose edges $\mathcal{E}_{SC_{acyclic}}$ is the largest subset of $\mathcal{E}_{SC}$ such that the graph is acyclic. $\mathcal{E}_{SC}$ is defined as

$$\{\ (T_{x_i},\, T_{x_j}) \mid (T_{x_i} <_{SC} T_{x_j}\ ) \wedge \neg(T_{x_i} <>_{CF} T_{x_j})\ \}$$

Next, one needs to find the connected components of an undirected graph whose nodes are $T_{x_1}$, ..., $T_{x_n}$ and whose edges are

$$\{\ (T_{x_i},\, T_{x_j}) \mid ((T_{x_i},\, T_{x_j}) \notin \mathcal{E}_{SC_{acyclic}}) \wedge ((T_{x_j},\, T_{x_i}) \notin \mathcal{E}_{SC_{acyclic}}) \wedge \neg(T_{x_i} <>_{CF} T_{x_j})\ \}$$

Each connected component forms a scheduling subgroup. Transitions in different scheduling subgroups are either conflict-free or sequentially-composible. $\phi$'s for the transitions in a subgroup can be generated using a priority encoder. On each clock cycle, the transitions $T_{y_1}$,...,$T_{y_m}$, selected collectively by all subgroup encoders of a particular scheduling group, satisfy the conditions of Theorem 3.3 because the selected transitions can always be ordered according to the static SC-ordering of the parent scheduling group.

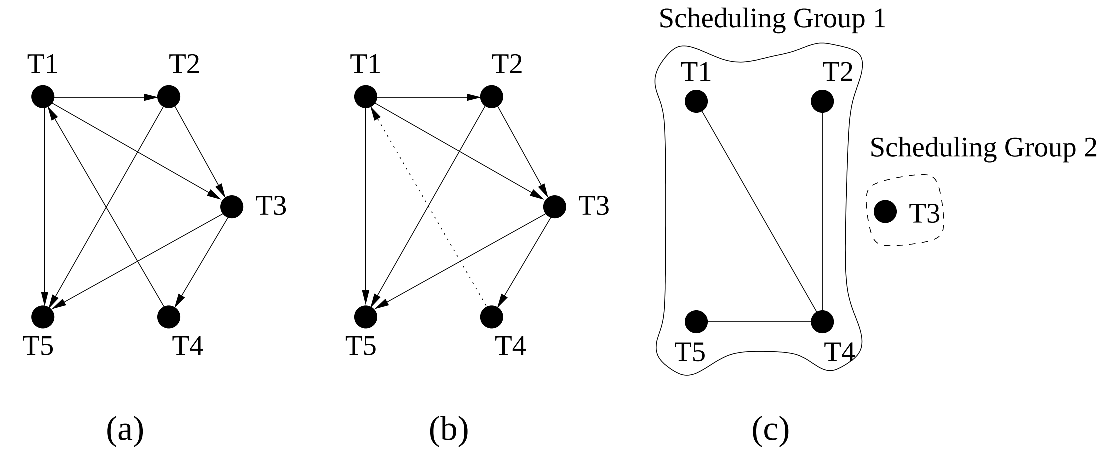


Figure 3-8: Scheduling sequentially-composible rules: (a) Directed $<_{SC}$ graph (b) Corresponding acyclic directed $<_{SC}$ graph (c) Corresponding conflict graph and its connected components.

**Example 3.8 (Sequentially Composed Implementation)**

The directed graph (a) in Figure 3-8 depicts the sequential composibility relationships in an ATS with five transitions, $\mathcal{X}=\{T_1, T_2, T_3, T_4, T_5\}$. A directed edge from $T_a$ to $T_b$ implies $T_a <_{SC} T_b$, i.e.,

$(T_1 <_{SC} T_2)$, $(T_1 <_{SC} T_3)$, $(T_1 <_{SC} T_5)$,
$(T_2 <_{SC} T_3)$, $(T_2 <_{SC} T_5)$,
$(T_3 <_{SC} T_4)$, $(T_3 <_{SC} T_5)$,
$(T_4 <_{SC} T_1)$

Graph (b) in Figure 3-8 shows the acyclic $<_{SC}$ graph when the edge from $T_4$ to $T_1$ is removed. A topological sort of the acyclic $<_{SC}$ graph yields the SC-ordering of $T_1$, $T_2$, $T_3$, $T_4$ and $T_5$. (The order of $T_4$ and $T_5$ can be reversed also.) Graph (c) in Figure 3-8 gives the corresponding conflict graph. The two connected components of the conflict graph are the two scheduling groups. $\phi_{T_3}=\pi_{T_3}$ without any arbitration. The remaining $\phi$ signals can be generated using a priority encoding of their corresponding $\pi$'s. More transitions can be executed concurrently if the enumerated encoder in Figure 3-9 is used instead.

□

**State Update Logic:**

When sequentially-composible transitions are allowed in the same clock cycle, the register update logic cannot assume that only one transition acts on a register in each clock cycle. When multiple actions are enabled for a register, the register update logic should ignore all except for the latest action with respect to the SC-ordering of some scheduling group. (All transitions that update the same register are in the same

| $\pi_{T_1}$ | $\pi_{T_2}$ | $\pi_{T_4}$ | $\pi_{T_5}$ | $\phi_{T_1}$ | $\phi_{T_2}$ | $\phi_{T_4}$ | $\phi_{T_5}$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |

Figure 3-9: Enumerated encoder table for Example 3.8.

scheduling group, except for a transition that is mutually exclusive with the rest.) For each R in $\mathcal{S}$, the set of transitions that update R is

$$\{\, T_{x_i} \mid \mathbf{a}^{R}_{T_{x_i}} = \boldsymbol{set}(exp_{x_i}) \,\}$$

The register's latch enable signal is

$$\mathsf{LE} = \phi_{T_{x_1}} \vee \ldots \vee \phi_{T_{x_n}}$$

The register's data input signal is

$$\mathsf{D} = \phi_{T_{x_1}} \cdot \psi_{T_{x_1}} \cdot exp_{x_1} + \ldots + \phi_{T_{x_n}} \cdot \psi_{T_{x_n}} \cdot exp_{x_n}$$

where $\psi_{T_{x_i}} = \neg(\phi_{T_{y_1}} \vee \phi_{T_{y_2}} \vee \ldots)$. The expression $\psi_{T_{x_i}}$ contains $\phi_{T_{y_i}}$'s from the set of transitions:

$$\begin{aligned} \{\ T_{y_i} \mid\ & \mathsf{R} \in \boldsymbol{R}(\alpha_{T_{y_i}}) \wedge \\ & T_{x_i} \text{ comes before } T_{y_i} \text{ in the } \textit{SC-ordering} \wedge \\ & \neg(T_{x_i} <>_{ME} T_{y_i})\ \} \end{aligned}$$

In essence, the register's data input (D) is selected through a prioritized multiplexer that gives higher priority to transitions later in the SC-ordering. The update logic for arrays and FIFOs remain unchanged from Section 3.2.3.

## 3.5 Summary

This chapter describes the procedure to synthesize an operation-centric description given as an ATS into an efficient synchronous digital implementation. The chapter first gives a straightforward implementation that executes one operation per clock cycle. The chapter next develops the necessary theories and algorithms to construct more optimized implementations that can carry out multiple operations concurrently while still maintaining a behavior that is consistent with a sequential execution of the atomic operations.

# Chapter 4

# TRSpec Hardware Description Language

The TRSpec language is an adaptation of Term Rewriting Systems (TRSs) [BN98, Klo92] for operation-centric hardware description. This synthesizable TRS language includes extensions beyond the standard TRS formalism to facilitate hardware description. On the other hand, the generality of TRS has to be restricted in some areas to ensure synthesizability. This chapter first introduces the basic concepts in the TRS formalism. Next, the chapter presents the TRSpec language. The discussion begins with TRSpec's type system and then moves onto the topics of rewrite rules, initialization and input/output. The concrete syntax of TRSpec is given separately in Appendix A. This chapter concludes with a discussion on how to map a TRSpec description to an ATS.

## 4.1 Term Rewriting Systems

TRSs have been used extensively to give operational semantics of programming languages. More recently, TRSs have become a powerful tool, in research and in classrooms, to concisely describe the operation of computer hardware. For example, TRSs have made it possible to describe a processor with out-of-order and speculative execution in a page of text [AS99]. Such hardware descriptions in TRSs are also amenable to formal verification techniques.

A TRS consists of a set of terms and a set of rewrite rules. The general structure of a rewrite rule is

$$\mathsf{pat}_{lhs}\ \textit{if}\ \mathsf{exp}_{pred} \rightarrow \mathsf{exp}_{rhs}$$

A rule can be applied to rewrite a term $\mathsf{s}$ if the rule's left-hand-side pattern $\mathsf{pat}_{lhs}$ matches $\mathsf{s}$ (or a subterm in $\mathsf{s}$) and the predicate $\mathsf{exp}_{pred}$ evaluates to true. A successful pattern match binds the variables in $\mathsf{pat}_{lhs}$ to the corresponding subterms in $\mathsf{s}$. When a rule is applied, the new term is constructed by replacing the whole or part of $\mathsf{s}$ that matched $\mathsf{pat}_{lhs}$ with the resulting term from evaluating $\mathsf{exp}_{rhs}$.

The effect of a rewrite is atomic, that is, the whole term is "read" in one step and if the rule is applicable then a new term is returned in the same step. If several rules are applicable, then any one of them is chosen nondeterministically and applied. Afterwards, all rules are re-evaluated for applicability on the new term. Starting from a *source* term, successive rounds of rewrites continue until the final term cannot be rewritten using any rule. TRSs that describe hardware are sometimes nondeterministic (*"not confluent"* in the programming language parlance) and nonterminating.

In a functional interpretation, a rule can be thought of as the function:

```
λ s. case s of
          pat_lhs ⇒ if exp_pred then
                          exp_rhs
                      else
                          s
          ... ⇒ s
```

This function uses a pattern-matching *case* statement in which two patterns, $\mathsf{pat}_{lhs}$ and '...', are checked sequentially against $\mathsf{s}$ until the first successful match. If $\mathsf{pat}_{lhs}$ matches $\mathsf{s}$, the variables in $\mathsf{pat}_{lhs}$ are bound to the subterms in $\mathsf{s}$, and the function returns the evaluation of the consequent expression $\mathsf{exp}_{rhs}$. If $\mathsf{pat}_{lhs}$ fails to match $\mathsf{s}$, the wild-card pattern '...' always matches $\mathsf{s}$ successfully, and the function returns a term identical to $\mathsf{s}$.

## 4.2 TRS for Hardware Description

Chapter 2 uses the TRS notation to describe the behaviors of finite state machines. In those descriptions, the collective values of the state elements are symbolically represented as terms. Each rewrite rule describes an atomic operation by specifying when the operation could take place and what is the new value of the state elements afterwards. In general, a TRS for hardware description has to be restricted to have a finite number of finite-size terms, and the rewrite rules cannot change the size of the terms. These restrictions guarantee that a TRS can be synthesized using a finite amount of hardware. These restrictions can be enforced by augmenting the TRS formalism with a type system.

The rest of this chapter presents TRSPEC, a strongly-typed TRS language for operation-centric hardware description. TRSPEC features built-in integers, common arithmetic and logical operators, non-recursive algebraic types and two built-in abstract datatypes, array and FIFO. User-defined abstract datatypes, with both sequential and combinational functionalities, can also be imported into a TRSPEC description by providing an interface declaration and a separate implementation description.

## 4.3 Type System

The TRSpec language is strongly typed, that is, every term has a type that is defined by the user. The allowed type classes in TRSpec are listed below.

$$
\begin{array}{rcl}
\mathsf{TYPE} & :: & \mathsf{STYPE} \\
 & [] & \mathsf{CPRODUCT} \\
 & [] & \mathsf{ABSTRACT} \\
 & [] & \mathsf{IOTYPE} \\
\mathsf{CPRODUCT} & :: & \mathsf{Cn}_k(\mathsf{TYPE}_1, ..., \mathsf{TYPE}_k)\ \textit{where}\ k > 0 \\
\mathsf{ABSTRACT} & :: & \mathsf{Array}\ [\mathsf{STYPE}_{idx}]\ \mathsf{STYPE} \\
 & [] & \mathsf{Fifo\ STYPE} \\
\mathsf{IOTYPE} & :: & \mathsf{ITYPE} \\
 & [] & \mathsf{OTYPE}
\end{array}
$$

### 4.3.1 Simple Types

Simple types (STYPE) in TRSpec include built-in booleans, signed and unsigned integers, as well as algebraic product and sum types. An integer type can be declared to have an arbitrary word width. Each product type has a unique *constructor name* and consists of one or more subtypes. A sum type, a disjoint union type, is made up of two or more disjuncts. A disjunct is similar to a product except a disjunct may have zero elements. A sum type with only zero-element disjuncts is an enumeration type. Product and sum types can be composed to construct arbitrary algebraic type structures, but recursive types are not allowed. Members of the STYPE class are listed below.

$$
\begin{array}{rcl}
\mathsf{STYPE} & :: & \mathsf{Bool} \\
 & [] & \mathsf{Bit}[N] \\
 & [] & \mathsf{Int}[N] \\
 & [] & \mathsf{PRODUCT} \\
 & [] & \mathsf{SUM} \\
\mathsf{PRODUCT} & :: & \mathsf{Cn}_k(\mathsf{STYPE}_1, ..., \mathsf{STYPE}_k)\ \textit{where}\ k > 0 \\
\mathsf{SUM} & :: & \mathsf{DISJUNCT} \mid\mid \mathsf{DISJUNCT} \\
 & [] & \mathsf{DISJUNCT} \mid\mid \mathsf{SUM} \\
\mathsf{DISJUNCT} & :: & \mathsf{Cn}_k(\mathsf{STYPE}_1, ..., \mathsf{STYPE}_k)\ \textit{where}\ k \geq 0
\end{array}
$$

**Example 4.1 (Euclid's Algorithm)**
The terms in the TRS from Section 2.1 has the type declared as GCD below.

$$
\begin{array}{llcl}
\mathsf{Type} & \mathsf{GCD} & = & \mathsf{Gcd(NUM,\ NUM)} \\
\mathsf{Type} & \mathsf{NUM} & = & \mathsf{Bit[32]}
\end{array}
$$

A GCD term fits the signature Gcd(a,b), where a and b are unsigned 32-bit integers.
□

**Example 4.2 (Alternate Description of Euclid's Algorithm)**

This example gives an alternate description of Euclid's Algorithm to illustrate some modularity and type-related issues. The following TRS describes the computation of the *mod* function via iterative subtraction.

```
Type   VAL  =  Mod(NUM,NUM) || Val(NUM)
Type   NUM  =  Bit[32]
```

*Mod Iterate Rule:*

```
    Mod(a,b) if a≥b → Mod(a-b,b)
```

*Mod Done Rule:*

```
    Mod(a,b) if a<b → Val(a)
```

Using this description of *mod*, Euclid's Algorithm can be rewritten as follows:

```
Type   GCD  =  Gcd(VAL,VAL)
```

*Flip & Mod Rule:*

```
    Gcd(Val(a), Val(b)) if b≠0 → Gcd(Val(b), Mod(a,b))
```

To find the greatest common divisor of two unsigned integers, $\mathsf{C}_a$ and $\mathsf{C}_b$, the source term should be Gcd(Val($\mathsf{C}_a$),Val($\mathsf{C}_b$)). The normal term is Gcd(Val($\mathsf{C}_{gcd}$),Val(0)) where $\mathsf{C}_{gcd}$ is the greatest common divisor of $\mathsf{C}_a$ and $\mathsf{C}_b$. The sequence of rewrites to find the GCD of 2 and 4 is

```
Gcd(Val(2),Val(4)) → Gcd(Val(4),Mod(2,4)) → Gcd(Val(4),Val(2)) →
Gcd(Val(2),Mod(4,2)) → Gcd(Val(2),Mod(2,2)) → Gcd(Val(2),Mod(0,2)) →
Gcd(Val(2),Val(0))
```

□

### 4.3.2 Abstract Types

Abstract datatypes are defined by their interfaces, without information about its internal operation or implementation. An interface can be classified as performing either a *combinational* operation or a *state-transforming* operation. TRSPEC has built-in abstract datatypes of arrays and FIFOs.

An array is used to model register files, memories, etc. An array term has only two operations, read and write. In a rule, if a is an array, a[idx] gives the value stored in the idx'th location of a, and a[idx:=v], a state-transforming "write", is an array identical to a except location idx has been changed to the value v. TRSPEC supports arrays of STYPE entries indexed by enumeration-type or integer-type indices.

FIFOs provide the primary means of communication between different modules and pipeline stages. The two main state-transforming operations on FIFOs are enqueue and dequeue. Enqueuing an element e to a FIFO q is written as q.enq(e) while dequeuing the first element from q is written as q.deq( ). An additional state-transforming interface q.clear( ) clears the contents of q. The combinational query interface q.first( ) returns the value of the oldest element in q. FIFOs are restricted to

```
Type   PROC   =  Proc(PC, RF, IMEM, DMEM)
Type     PC   =  Bit[16]
Type     RF   =  Array [RNAME] VAL
Type  RNAME   =  Reg0( ) || Reg1( ) || Reg2( ) || Reg3( )
Type    VAL   =  Bit[16]
Type   IMEM   =  Array [PC] INST
Type   INST   =  Loadi(RNAME, VAL)
              || Loadpc(RNAME)
              || Add(RNAME, RNAME, RNAME)
              || Sub(RNAME, RNAME, RNAME)
              || Bz(RNAME, RNAME)
              || Load(RNAME, RNAME)
              || Store(RNAME, RNAME)
Type   DMEM   =  Array [ADDR] VAL
Type   ADDR   =  Bit[16]
```

Figure 4-1: Type definitions for a simple non-pipelined processor.

have STYPE entries. In the description phase, a FIFO is abstracted to have a finite but unspecified size. A TRSpec rule that uses a FIFO's interfaces has an implied predicate that tests whether the FIFO is not empty or not full, as appropriate.

TRSpec allows abstract types to be included in the construction of new product types, but an abstract type cannot be included in the algebraic type hierarchy descending from a sum type. Hence, TRSpec distinguishes between a complex product type (CPRODUCT) versus a simple product type (PRODUCT), depending on whether or not its algebraic structure contains an abstract type. Complex product types cannot be included in the disjunct of a sum type.

**Example 4.3 (Simple Processor)**

The TRS in Section 2.2 describes a simple ISA whose programmer-visible states consist of a program counter, a register file, instruction memory and data memory. The terms from that TRS have the types declared in Figure 4-1. A term that captures the processor state has the type PROC. Instruction terms have the type INST.

□

**Example 4.4 (Pipelined Processor)**

The TRS in Section 2.4 describes a two-stage pipelined processor. The processor performs instruction fetch and decode in the first stage and instruction execute in the second stage. A FIFO stores decoded instruction templates between the two pipeline stages. A term that captures the state of this pipelined processor has the type:

```
Type   PROC   =   Proc(PC, RF, BF, IMEM, DMEM)
```

The type of the abstract FIFO (BF) and the type of the instruction templates (ITEMP) are defined as

```
Type       BF   =   Fifo ITEMP
Type   ITEMP   =   TLoadi(RNAME, VAL)
                ||  TAdd(RNAME, VAL, VAL)
                ||  TSub(RNAME, VAL, VAL)
                ||  TBz(VAL, PC)
                ||  TLoad(RNAME, ADDR)
                ||  TStore(ADDR, VAL)
```

□

## 4.4 Rewrite Rules

### 4.4.1 Abstract Syntax

Syntactically, a TRSpec rewrite rule is composed of a left-hand-side pattern and a right-hand-side expression. A rule can optionally include a predicate expression and *where* bindings. The *where* bindings on the left-hand-side have pattern-matching semantics. A failure in matching $\mathsf{PAT}_i$ to $\mathsf{EXP}_{lhs,i}$ in the $i$'th left-hand-side binding also deems the rule inapplicable to a term. The right-hand-side *where* bindings are simple irrefutable variable assignments. The concrete TRSpec syntax is given in Appendix A. The abstract syntax of a TRSpec rewrite rule is summarized below. ('–' is the don't-care symbol.)

| | | |
|---|---|---|
| RULE | :: | LHS $\rightarrow$ RHS |
| LHS | :: | $\mathsf{PAT}_{lhs}$ [*if* $\mathsf{EXP}_{pred}$] [*where* $\mathsf{PAT}_1$=$\mathsf{EXP}_{lhs,1}$,...,$\mathsf{PAT}_n$=$\mathsf{EXP}_{lhs,n}$] |
| PAT | :: | '–' [] *variable* [] numerical constant [] $\mathsf{Cn}_0$( ) [] $\mathsf{Cn}_k$($\mathsf{PAT}_1$,...,$\mathsf{PAT}_k$) |
| RHS | :: | $\mathsf{EXP}_{rhs}$ [*where* *variable*$_1$=$\mathsf{EXP}_{rhs,1}$,...,*variable*$_n$=$\mathsf{EXP}_{rhs,m}$] |
| EXP | :: | '–' [] *variable* [] numerical constant [] $\mathsf{Cn}_0$( ) [] $\mathsf{Cn}_k$($\mathsf{EXP}_1$,...,$\mathsf{EXP}_k$) |
| | [] | Primitive-Op ($\mathsf{EXP}_1$,...,$\mathsf{EXP}_k$) |
| Primitive-Op | :: | *Arithmetic* [] *Logical* [] *Array-Access* [] *FIFO-Access* |

### 4.4.2 Type Restrictions

The type of $\mathsf{PAT}_{lhs}$ must be PRODUCT, CPRODUCT or SUM. In addition, each rule must have $\mathsf{PAT}_{lhs}$ and $\mathsf{EXP}_{rhs}$ of the same type. In Example 4.2, the *Mod Done* rule rewrites a Mod term to a Val term. The *Mod Done* rule does not violate this type discipline because both a Val term and a Mod term belong to the same sum type.

TRSPEC's non-recursive type system and the type disciplines on rewrite rules ensure the size of every term is finite and a rewrite does not change the size of the terms.

### 4.4.3 Semantics

In a functional interpretation, a TRSPEC rewrite rule can be thought of as a function of TYPEOF($\mathsf{PAT}_{lhs}$)$\rightarrow$TYPEOF($\mathsf{PAT}_{lhs}$). The function returns a term identical to the input term if the rule is not applicable. If the rule is applicable, the return value is a new term based on the evaluation of the main right-hand-side expression $\mathsf{EXP}_{rhs}$. A rule of the form:

```
     $pat_{lhs}$
         $if$ $exp_{pred}$ $where$ $pat_1 = exp_{lhs,1}$, ..., $pat_n = exp_{lhs,n}$
  →  $exp_{rhs}$
         $where$ $var_1 = exp_{rhs,1}$, ..., $var_m = exp_{rhs,m}$
```

corresponds to the function:

```
$\lambda$ s. case s of
        $pat_{lhs} \Rightarrow$
            case $exp_{lhs,1}$ of
                $pat_1 \Rightarrow$
                    ......
                        case $exp_{lhs,n}$ of
                            $pat_n \Rightarrow$
                                if $exp_{pred}$ then
                                    let
                                        $var_1 = exp_{rhs,1}$, ..., $var_m = exp_{rhs,m}$
                                    in
                                        $exp_{rhs}$
                                else
                                    s
                            ... $\Rightarrow$ s
                    ......
                ... $\Rightarrow$ s
        ... $\Rightarrow$ s
```

The previous function can be decomposed into its two components, $\pi$ and $\delta$, where

```
$\pi = \lambda\ \mathsf{s}.\ \textit{case}\ \mathsf{s}\ \textit{of}$
            $\mathsf{pat}_{lhs} \Rightarrow$
                $\textit{case}\ \mathsf{exp}_{lhs,1}\ \textit{of}$
                    $\mathsf{pat}_1 \Rightarrow$
                        ......
                            $\textit{case}\ \mathsf{exp}_{lhs,n}\ \textit{of}$
                                $\mathsf{pat}_n \Rightarrow \mathsf{exp}_{pred}$
                                $... \Rightarrow \textit{false}$
                        ......
                    $... \Rightarrow \textit{false}$
            $... \Rightarrow \textit{false}$
```

and

```
$\delta = \lambda\ \mathsf{s}.\ \textit{let}$
            $\mathsf{pat}_{lhs} = \mathsf{s}$
            $\mathsf{pat}_1 = \mathsf{exp}_{lhs,1},\ ...,\ \mathsf{pat}_n = \mathsf{exp}_{lhs,n}$
            $\mathsf{var}_1 = \mathsf{exp}_{rhs,1},\ ...,\ \mathsf{var}_m = \mathsf{exp}_{rhs,m}$
        $\textit{in}$
            $\mathsf{exp}_{rhs}$
```

The $\pi$ function determines a rule's applicability to a term and has the type signature TYPEOF($\mathsf{pat}_{lhs}$)$\rightarrow$Bool. On the other hand, the $\delta$ function, with the type signature TYPEOF($\mathsf{pat}_{lhs}$)$\rightarrow$TYPEOF($\mathsf{pat}_{lhs}$), determines the new term in case $\pi(\mathsf{s})$ evaluates to true. Using $\pi$ and $\delta$, a rule can also be written as the function:

$\lambda\ \mathsf{s}.\ \textit{if}\ \pi(\mathsf{s})\ \textit{then}\ \delta(\mathsf{s})\ \textit{else}\ \mathsf{s}$

## 4.5 Source Term

A TRSpec description includes a source term to specify the initial state of a system. The top-level type of the system is inferred from the type of the source term, which must be PRODUCT, CPRODUCT or SUM. A source term can specify an initial value for every subterm that is not an ABSTRACT type. A source term can include don't-care symbols to leave some subterms unconstrained. Unless otherwise specified, the initial contents of an abstract array term are undefined. An abstract FIFO term is initially empty.

## 4.6 Input and Output

Traditionally, a TRS models a closed system, that is, it does not interact outside of the modeled system. TRSpec augments the TRS formalism to allow users to assign I/O semantics to STYPE terms. For example, the TRSpec fragment in Figure 4-2 can

```
Type    TOP    =  Top(CNTRI, NUMI, NUMI, NUMO, GCD)
IType   CNTRI  =  CNTR
Type    CNTR   =  Load( ) || Run( )
IType   NUMI   =  NUM
OType   NUMO   =  NUM

GCD Start Rule:
     Top(Load( ), x, y, –, –)
→    Top(Load( ), x, y, 0, Gcd(Val(x),Val(y)))
GCD Done Rule:
     Top(Run( ), x, y, –, Gcd(Val(gcd), Val(0)))
→    Top(Run( ), x, y, gcd, Gcd(Val(gcd), Val(0)))
```

Figure 4-2: TRSpec description of I/O

be combined with the TRSpec design from Example 4.2 to provide I/O capabilities. When synthesized as hardware, this I/O wrapper description maps to a module with the I/O ports shown in Figure 4-3.

TOP is a collection of several I/O types along with GCD. NUMO is an output type derived from NUM. An output type can be derived from any STYPE, and the output type itself can be included in the construction of new CPRODUCT types. In the implementation of a TRSpec description, the value of an output term is externally visible.

On the other hand, NUMI is an input type derived from NUM. Like an output type, an input type can also be derived from any STYPE and be included in the construction of new CPRODUCT types. The value of an input term can only be rewritten by an agent external to the system. The effect of an externally-triggered rewrite appears spontaneously but atomically between the atomic effects of rewrite rules. Rewrite rules inside the system are not allowed to change the value of an input term. In the implementation of a TRSpec description, input interfaces are created to control the values of the input terms.

Although the TRSpec compiler has some liberty in the exact implementation of the input and output interfaces, as a rule of thumb, one should be able to connect the output port from a correctly implemented module to the input port of another correctly implemented module, as long as the ports are derived from the same type.

The *GCD Start* and *GCD Done* rules describe the input and output operations. Given a TOP term, the *GCD Start* rule states that when the first subterm is Load( ), the GCD subterm can be initialized using the second and third subterms. (The first three subterms of a TOP term are all input terms.) The *GCD Done* rule states if the first subterm is Run( ) and the GCD subterm looks like Gcd(Val(a),Val(0)) then the

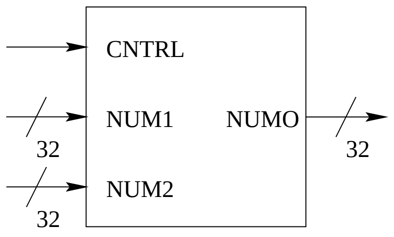


Figure 4-3: I/O interfaces synthesized for TOP.

NUMO output subterm should be set to a to report the answer.

Using the symbol '$\hookrightarrow$' to represent externally-triggered rewrites, a possible sequence of rewrites that computes the GCD of 2 and 4 is

.... $\rightarrow$Top(Run(),9,3,3,Gcd(Val(3),Val(0)))
$\hookrightarrow$Top(Load(),2,4,3,Gcd(Val(3),Val(0))) $\rightarrow$Top(Load(),2,4,0,Gcd(Val(2),Val(4)))
$\hookrightarrow$Top(Run(),2,4,0,Gcd(Val(2),Val(4))) $\rightarrow$Top(Run(),2,4,0,Gcd(Val(4),Mod(2,4)))
$\rightarrow$Top(Run(),2,4,0,Gcd(Val(4),Val(2))) $\rightarrow$Top(Run(),2,4,0,Gcd(Val(2),Mod(4,2)))
$\rightarrow$Top(Run(),2,4,0,Gcd(Val(2),Mod(2,2))) $\rightarrow$Top(Run(),2,4,0,Gcd(Val(2),Mod(0,2)))
$\rightarrow$Top(Run(),2,4,0,Gcd(Val(2),Val(0))) $\rightarrow$Top(Run(),2,4,2,Gcd(Val(2),Val(0)))
$\hookrightarrow$ ....

At the start of the sequence, the first external rewrite modifies the first three subterms of Top(Run( ),9,3,3,Gcd(Val(3),Val(0))) to Top(Load( ),2,4,3,Gcd(Val(3),Val(0))). Afterwards, the *GCD Start* rule is applied to set the GCD subterm to Gcd(Val(2),Val(4)). Another external rewrite changes the first subterm from Load( ) to Run( ); this disables the *GCD Start* rule. Applications of the rules from Example 4.2 ultimately reduce the GCD subterm from Gcd(Val(2),Val(4)) to Gcd(Val(2),Val(0)). At this point, the *GCD Done* rule becomes enabled and rewrites the NUMO subterm from 0 to 2.

Due to nondeterminism, many sequences are possible for the same external manipulations of the input subterms. Therefore, I/O handshakes in a TRSpec description must be asynchronous by design. For example, after the first external rewrite there is no guarantee on how soon the *GCD Start* rule is applied. Hence, the external agents should not trigger the second external rewrite until it has observed a transition to 0 on the NUMO output. Also before the second external rewrite, the *GCD Start* rule could be applied several more times. Moreover, the GCD rules from Example 4.2 could also start rewriting the GCD subterm. Nevertheless, regardless of the sequence of events between the first and the second external rewrites, this TRS produces the correct answer when the NUMO output term transitions out of 0.

## 4.7 Mapping TRSpec to ATS

The TRSpec type discipline requires each rewrite rule to have the same type on both sides of $\rightarrow$. Therefore, all terms reachable from the source term by rewriting must have the same type. Given TRSpec's non-recursive type system, the terms in

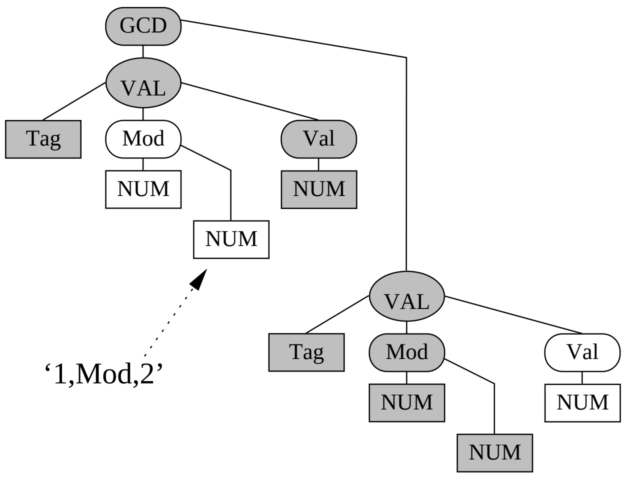


Figure 4-4: A tree graph representing the GCD type structure from Example 4.2.

a TRSpec TRS can always be encoded as the collective value of a finite set of ATS state elements.

## 4.7.1 Terms to ATS State Elements

The ATS state elements needed to represent all possible terms of the same type as the source term can be conceptually organized in a tree graph according to the source term's algebraic type structure. In this tree, the ATS state elements, R, A and F, appear at the leaves. For example, the tree graph of the algebraic type, GCD, from Example 4.2 is shown in Figure 4-4. A product node, like the one labeled GCD, has a child branch for each constituent subtype. A sum node, like the one labeled VAL, has a branch for each disjunct. A sum node also has an extra branch where a $\lceil log_2 d \rceil$-bit "tag register" node records which one of the $d$ disjuncts is active. A disjunct node, like the one labeled Mod, has a child branch for each of its constituent subtypes. In this case, the Mod disjunct node has two leaf children that are 32-bit register nodes corresponding to the 32-bit integer type NUM.

A sum node has a branch for each of the disjuncts, but, at any time, only the branch whose tag matches the content of the tag register holds meaningful data. For example, the active subtree corresponding to the term Gcd(Val(2), Mod(4,2)) is shaded in Figure 4-4. As an optimization, registers on different disjuncts of a sum node can share the same physical register. In Figure 4-4, the registers aligned horizontally can be allocated to the same physical register.

A unique name can be assigned to each ATS state element based on the path (also known as projection) from the root to the corresponding leaf node. For example, the name for the second (from the left) NUM register in Figure 4-4 is '1, Mod, 2'. Following this naming scheme, the ATS state elements needed by a type can also be organized as a set of name/state-element pairs where each pair $\langle$ *proj*, *e* $\rangle$ specifies an ATS state element *e* and its name *proj*.

The storage elements (and their names) needed to represent all possible term instances of GCD are given by the set:

$$
\begin{array}{l}
\{\ \langle\ \text{`}1, \mathit{tag}\text{'},\ \mathsf{R}_{1-\mathit{bit}}\ \rangle, \\
\qquad\quad \langle\ \text{`}1, \mathsf{Mod}, 1\text{'},\ \mathsf{R}_{32-\mathit{bit}}\ \rangle,\ \langle\ \text{`}1, \mathsf{Mod}, 2\text{'},\ \mathsf{R}_{32-\mathit{bit}}\ \rangle,\ \langle\ \text{`}1, \mathsf{Val}, 1\text{'},\ \mathsf{R}_{32-\mathit{bit}}\ \rangle, \\
\quad \langle\ \text{`}2, \mathit{tag}\text{'},\ \mathsf{R}_{1-\mathit{bit}}\ \rangle, \\
\qquad\quad \langle\ \text{`}2, \mathsf{Mod}, 1\text{'},\ \mathsf{R}_{32-\mathit{bit}}\ \rangle,\ \langle\ \text{`}2, \mathsf{Mod}, 2\text{'},\ \mathsf{R}_{32-\mathit{bit}}\ \rangle,\ \langle\ \text{`}2, \mathsf{Val}, 1\text{'},\ \mathsf{R}_{32-\mathit{bit}}\ \rangle\ \}
\end{array}
$$

Every name/state-element pair in this set has a one-to-one correspondence with a leaf node in the tree graph in Figure 4-4.

### 4.7.2 Rules to ATS Transitions

With a nearly identical execution semantics as an ATS transition, a TRSPEC rewrite rule readily maps to an ATS transition. Using the functional interpretation of rules presented in Section 4.4.3, the $\pi$ and $\delta$ functions for the *Flip&Mod* rule from Example 4.2 are

$$
\begin{array}{l}
\pi = \lambda\ \mathsf{s}.\ \mathit{case}\ \mathsf{s}\ \mathit{of} \\
\qquad\qquad\qquad \mathsf{Gcd(Val(a),Val(b))} \Rightarrow \mathsf{b{\neq}0} \\
\qquad\qquad\qquad ... \Rightarrow \mathit{false}
\end{array}
$$

and

$$
\begin{array}{l}
\delta = \lambda\ \mathsf{s}.\ \mathit{let} \\
\qquad\qquad\quad \mathsf{Gcd(Val(a),Val(b))} = \mathsf{s} \\
\qquad\qquad \mathit{in} \\
\qquad\qquad\quad \mathsf{Gcd(Val(b),Mod(a,b))}
\end{array}
$$

The $\pi$ function of a rule maps to the $\pi$ expression of an ATS transition. In an ATS, the condition computed by $\pi(\mathsf{s})$ is specified as a logic expression that tests the contents of the ATS state elements. The $\pi$ expression implied by the pattern matching and the predicate of the *Flip&Mod* rule is

$$(\mathsf{R}_{1,tag} = \mathsf{Val}) \wedge (\mathsf{R}_{2,tag} = \mathsf{Val}) \wedge (\mathsf{R}_{2,Val,1} \neq 0)$$

The $\delta$ function of a rule can be viewed as specifying actions on the ATS state elements. The $\delta$ function of the *Flip&Mod* rule can be mapped to the following set of state-element/action pairs, $\langle\ \mathsf{e},\ \mathsf{action}\ \rangle$. (Elements with null actions are omitted.)

$$
\begin{array}{l}
\{\ \langle\ \mathsf{R}_{1,tag},\ \boldsymbol{set}(\mathsf{Val})\ \rangle,\ \langle\ \mathsf{R}_{1,\mathsf{Val},1},\ \boldsymbol{set}(\mathrm{b})\ \rangle, \\
\qquad \langle\ \mathsf{R}_{2,tag},\ \boldsymbol{set}(\mathsf{Mod})\ \rangle,\ \langle\ \mathsf{R}_{2,\mathsf{Mod},1},\ \boldsymbol{set}(\mathrm{a})\ \rangle,\ \langle\ \mathsf{R}_{2,\mathsf{Mod},2},\ \boldsymbol{set}(\mathrm{b})\ \rangle\ \ \}
\end{array}
$$

The variables $\mathsf{a}$ and $\mathsf{b}$ in the *Flip&Mod* rule's $\delta$ function are bound to the subterms in the input term $\mathsf{s}$ by pattern matching. In the ATS context, $\mathsf{a}$ and $\mathsf{b}$ are the contents of the corresponding ATS storage elements, namely $\mathsf{R}_{1,Val,1}$ and $\mathsf{R}_{2,Val,1}$.

Notice, the $\pi$ expression of this transition requires $\mathsf{R}_{1,tag}$=$\mathsf{Val}$. Thus, the action

$\langle\, \mathsf{R}_{1,tag},\ \boldsymbol{set}(\mathsf{Val})\,\rangle$ in the set above is redundant in any state that satisfies the transition's predicate. The ***set*** action on $\mathsf{R}_{1,tag}$ can be replaced by a null action without changing the semantics of the transition. In general, the predicate of a transition restricts the possible starting values of the state when the transition is applied. A compiler can eliminate an action from a transition if the compiler can statically detect that the action has no effect on its state element anytime the transition's predicate is satisfied.

In another example, the pipelined processor TRS from Example 4.4 requires the following set of storage elements:

$$\{\ \langle\, \text{`1'},\ \mathsf{R}_{PC}\,\rangle,\ \langle\, \text{`2'},\ \mathsf{A}_{RF}\,\rangle,\ \langle\, \text{`3'},\ \mathsf{F}_{BF}\,\rangle,\ \langle\, \text{`4'},\ \mathsf{A}_{IMEM}\,\rangle,\ \langle\, \text{`5'},\ \mathsf{A}_{DMEM}\,\rangle\ \}$$

The transition corresponding to the *Fetch* rule has the following actions:

$$\{\ \langle\, \mathsf{R}_{PC},\ \boldsymbol{set}(\mathsf{R}_{PC}{+}1)\,\rangle,\ \langle\, \mathsf{F}_{BF},\ \boldsymbol{enq}(\mathsf{A}_{IMEM}[\mathsf{R}_{PC}])\,\rangle\ \}$$

The transition corresponding to the *Add Execute* rule has the following actions:

$$\{\ \langle\, \mathsf{A}_{RF},\ \boldsymbol{a\text{-}set}(\mathsf{A}_{RF}[\mathsf{r1}]{+}\mathsf{A}_{RF}.[\mathsf{r2}])\,\rangle,\ \langle\, \mathsf{F}_{BF},\ \boldsymbol{deq}(\,)\,\rangle\ \}$$

### 4.7.3 Local Rules

A local rule may be applicable to many different parts of the top-level term. In these cases, a local rule maps to multiple ATS transitions. It is as if a local rule is replaced by multiple lifted instances of the rule, one for each possible application site. The effect of applying a particular lifted instance to the whole term is the same as applying the original local rule to the corresponding local site. For example, the *Mod Done* rule from Example 4.2 can be applied to both the first and second subterms of a GCD term. The two lifted instances of the *Mod Done* rule are

$$\mathsf{Gcd(Mod(a,b),x)}\ \textit{if}\ \mathsf{a{<}b} \rightarrow \mathsf{Gcd(Val(a),x)}$$

and

$$\mathsf{Gcd(x,Mod(a,b))}\ \textit{if}\ \mathsf{a{<}b} \rightarrow \mathsf{Gcd(x,Val(a))}$$

## 4.8 Summary

This chapter presents the TRSpec operation-centric hardware description language. The TRSpec language is an adaptation of the TRS notation to enable concise and precise descriptions of hardware behavior. The TRSpec language includes both extensions and restrictions on the standard TRS formalism to increase its usefulness in hardware descriptions. The concrete syntax of TRSpec is given in Appendix A. This chapter also shows how to translate a TRSpec description to an equivalent ATS.

# Chapter 5

# Examples of TRSpec Descriptions and Synthesis

The procedures for synthesizing operation-centric descriptions, described in Chapter 3, has been implemented in the Term Rewriting Architectural Compiler (TRAC). TRAC accepts TRSPEC descriptions and outputs synthesizable structural descriptions in the Verilog Hardware Description Language [TM96]. This chapter presents the results from applying TRAC to TRSPEC examples. To produce the final implementations, the TRAC-generated Verilog descriptions are further compiled by commercial hardware compilers to target a number of implementation technologies. The quality of TRAC-generated implementations is evaluated against reference implementations synthesized from hand-coded Verilog structural descriptions.

## 5.1 Euclid's Algorithm

TRSPEC and TRAC combine to form a high-level hardware development framework. Staying within the TRS formalism, even someone without a digital design background can capture an algorithm and produce a hardware implementation.

### 5.1.1 Design Capture

The TRSPEC language offers a level of abstraction and conciseness that is comparable to high-level programming languages. Example 4.2 gives a description of Euclid's Algorithm as a TRS. The description is reiterated in Figure 5-1 using concrete TRSPEC syntax. This TRSPEC description compares favorably against a software implementation of the same algorithm in Scheme (shown in Figure 5-2). In both TRSPEC and Scheme, a designer/programmer can rely on the high-level language abstractions to express the algorithm in a direct and intuitive way. Notice that both the TRSPEC and the Scheme descriptions are easy to understand even without comments.

In comparison, a hand-coded Verilog register-transfer level (RTL) description of Euclid's Algorithm (excerpt shown in Figure 5-3) cannot be created or understood by someone who is not familiar with synchronous logic design. Besides the information

```
Type  GCD  =  Gcd(VAL, VAL)
Type  VAL  =  Mod(NUM, NUM) || Val(NUM)
Type  NUM  =  Bit[32]

Rule "Flip & Mod"
  Gcd(Val(a), Val(b)) if b!=0 ==> Gcd(Val(b), Mod(a,b))

Rule "Mod Iterate"
  Mod(a,b) if a>=b ==> Mod(a-b,b)

Rule "Mod Done"
  Mod(a,b) if a<b ==> Val(a)
```

Figure 5-1: TRSpec description of Euclid's Algorithm.

inherent to the algorithm, an RTL designer must also inject information about synchronous hardware implementation, such as exactly how the algorithm is scheduled over multiple clock cycles.

### 5.1.2 Debugging

As discussed in Section 2.1, the correctness of a TRSpec description can be established by verifying the correctness of each rewrite rule independently. Each rule only has to individually maintain the invariant that given any valid term, if enabled, the rule must produce another valid term. TRAC guarantees that the synthesized implementation behaves according to some valid sequence of atomic rewrites, thus producing a corresponding sequence of valid states. In practice, TRAC also helps designers uncover many mistakes by type checking the TRSpec sources.

Once the correctness of each individual rule is verified, the remaining mistakes are likely to be either due to unintended nondeterminism or missing rules. Executing a TRS with unintended nondeterminism can result in a livelock where the rewrite sequence repeats without making progress. On the other hand, the rewrite sequence of a TRS with an incomplete set of rules can terminate prematurely. The TRAC compiler can assist in debugging these two classes of errors by generating a simulatable Verilog description. The simulatable description has the same state structure and cycle-by-cycle behavior as the synthesizable description. However, the simulatable description can be instrumented to print an on-screen warning, or halt the simulation, whenever multiple rules are enabled in the same clock cycle. A designer can then verify if the nondeterminism exists by design. Likewise, the simulatable Verilog description can also issue a warning when none of the rules are enabled in a clock cycle. The designer can examine the state of a stopped simulation and determine if new rules

```
(define (gcd a b)
  (if (= b 0)
    a
    (gcd b (iter-remainder a b))))

(define (iter-remainder a b)
  (if (< a b)
    a
    (iter-remainder (- a b) b)))
```

Figure 5-2: Scheme implementation of Euclid's Algorithm.

are needed to complete the description.

### 5.1.3 Synthesis Results

The algorithmic description in Figure 5-1 needs to be combined with the TRSpec description of a top-level I/O wrapper (similar to the example in Section 4.6) to form a usable hardware module. TRAC can compile the combined TRSpec description into a synthesizable Verilog RTL description in less than 2 seconds on a workstation with a 500 MHz Pentium III processor. The TRAC-generated Verilog description can then be compiled for implementation on a Xilinx XC4010XL-09 FPGA [Xila] using the Xilinx Foundation 1.5i tools [Xilb]. For this example, the compilation time in the Xilinx tools is about 10 minutes.

The table in Figure 5-4 reports the circuit characteristics of the synthesized FPGA implementation in terms of the number of flip-flops, the overall FPGA resource utilization, and the maximum clock frequency. The row labeled "TRSpec" in Figure 5-4 characterizes the implementation synthesized from the TRSpec description. For comparison, an implementation is also synthesized from the hand-coded Verilog RTL description (Figure 5-3); its circuit characteristics are reported in the row labeled "Hand-coded RTL". To account for the algorithmic differences between the two designs, the last column of the table reports the number of clock cycles needed to compute the GCD of 590,111,149 and 998,829,163.[1]

The results indicate that the implementation synthesized from the TRSpec description is much worse than the version synthesized from the hand-coded Verilog RTL description. In comparison to the hand-coded RTL design, the TRAC-synthesized design requires 38 additional bits of storage, has a longer critical path, and needs an extra cycle for every remainder computed. This difference in circuit quality and

[1] 590111149=53857×10957 and 998829163=91159×10957. 53857, 10957 and 91150 are prime numbers.

```
module GCD (
   Gcd, Done, Start,       // Outputs
   Clk, Reset, Mode, A, B  // Inputs
);

   . . . . . . . .

   reg [0:31]    x;
   reg [0:31]    y;

   . . . . . . . .

   always @ (posedge Clk or negedge Reset)
     begin
        if (!Reset) begin
             x <= 0;
             y <= 0;
        end else if (newStart) begin
             x <= A;
             y <= B;
        end else if (x >= y) begin
             x <= x - y;
             y <= y;
        end else begin
             x <= y;
             y <= x;
        end
     end

   . . . . . . . .

endmodule
```

*(Courtesy of D. L. Rosenband.)*

Figure 5-3: Excerpt from the hand-coded Verilog RTL description of Euclid's Algorithm.

| Version | FF (bit) | Util. (%) | Freq. (MHz) | Elapse (cyc) |
|---|---|---|---|---|
| TRSpec | 102 | 38 | 31.5 | 104 |
| Hand-coded RTL | 64 | 16 | 53.1 | 54 |
| TRSpec (Optimized) | 64 | 20 | 44.2 | 54 |

Figure 5-4: Summary of GCD synthesis results.

performance is a consequence of the trade-off between performance and ease of development in a high-level design framework. In creating a hand-coded RTL description, a human designer needs to do extra work to inject some minimal amount of implementation-related information, but, at the same time, the RTL designer can also inject "creative" optimizations to achieve a smaller and simpler implementation. TRAC does not have the same ingenuity; after all, TRAC can only synthesize what has been described and cannot invent new mechanisms. For simple designs, a human designer can almost always create a better circuit directly in a lower-level abstraction than relying on automatic synthesis from a high-level abstraction. However, as the design complexity increases, the benefit of low-level hand optimizations quickly diminishes while the required effort increases dramatically. The advantage of high-level design and synthesis becomes more prominent in larger designs where managing complexity and concurrency becomes the dominant problem. This trend is already evident in the development of a simple processor in the next section.

Although TRAC cannot match the ingenuity of a human designer, the TRSpec language does not prevent a performance-conscious designer from adding source-level optimizations to a description. With an understanding of how TRAC maps a TRSpec description to its RTL implementation, a designer can influence the synthesized outcome by making changes to the source-level description. Figure 5-5 gives another TRSpec description of Euclid's Algorithm that has been formulated to match the hand-coded RTL design. (The rules and type definitions in Figure 5-5 are first shown in Section 2.1 and Example 4.1.) The implementation of the optimized TRSpec description shows a large improvement over the more literal algorithmic-style description. The implementation synthesized from the optimized TRSpec description is characterized in the row labeled "TRSpec (Optimized)" in Figure 5-4. The source-level optimizations result in an implementation that is within 25% of the hand-coded RTL design in terms of overall circuit size and 17% in terms of circuit speed.

This design example serves to demonstrate the TRSpec design flow for implementing an algorithm in hardware and also to compare the TRAC-synthesized implementations against traditional hand implementations. One caveat of this example is that a software implementation of Euclid's Algorithm on the latest microprocessor

```
Type  GCD  =  Gcd(NUM, NUM)
Type  NUM  =  Bit[32]

Rule "Mod"
  Gcd(a,b) if (a>=b) && (b!=0) ==> Gcd(a-b,b)

Rule "Flip"
  Gcd(a,b) if (a<b) ==> Gcd(b,a)
```

Figure 5-5: Another TRSpec description of Euclid's Algorithm.

would, in fact, run faster than any FPGA implementation, provided the software implementation can directly compute the *mod* function on the processor's floating-point unit. For algorithm synthesis, the TRSpec framework is better suited for problems with a high degree of fine-grain concurrency and bit-level operations.

## 5.2 A Simple Processor

The rules and type definitions from Section 2.2 and Example 4.3 describe a simple Instruction Set Architecture (ISA). As an architectural description, it is convenient to model the instruction memory and the data memory as abstract arrays internal to the system. As is, the description can be synthesized to a processor with an internal instruction ROM and an internal data RAM. However, in a realistic design, the processor module should access external memory modules through input and output ports.

### 5.2.1 Adaptation for Synthesis

A new processor description in concrete TRSpec syntax is given in Figures 5-6 and 5-7. The rules and type definitions are derived directly from the architectural-level ISA description of Section 2.2 and Example 4.3. The new description synthesizes to a processor with instruction and data memory interfaces shown in Figure 5-8.

**Instruction Memory Interface:**

In the new definition of `PROC`, internal memory arrays have been replaced by I/O subterms. (The semantics of I/O types is described in Section 4.6.) `PC_O` is an output type derived from `PC`. The `PC_O` subterm in a `PROC` term is the program counter. Its value appears on a corresponding output port of the synthesized processor. `INSTPORT` is an input type derived from `INST`. The value of the `INSTPORT` subterm in a `PROC` term is taken from a corresponding input port of the synthesized processor. The

```
Type       PROC    =  Proc(PC_O, RF, INSTPORT, RPORT, WPORT)
OType      PC_O    =  PC
Type         PC    =  Bit[16]
Type         RF    =  Array [RNAME] VAL
Type      RNAME    =  Reg0() || Reg1() || Reg2() || Reg3()
Type        VAL    =  Bit[16]
IType  INSTPORT    =  INST
Type       INST    =  Loadi(RNAME,VAL)
                   || Loadpc(RNAME)
                   || Add(RNAME,RNAME,RNAME)
                   || Sub(RNAME,RNAME,RNAME)
                   || Bz(RNAME,RNAME)
                   || Load(RNAME,RNAME)
                   || Store(RNAME,RNAME)
Type      RPORT    =  Rport(ADDR_O, VAL_I, BUSY)
OType    ADDR_O    =  ADDR
Type       ADDR    =  Bit[16]
IType     VAL_I    =  VAL
Type       BUSY    =  Busy() || NotBusy()
Type      WPORT    =  Wport(ADDR_O, VAL_O, ENABLE_O)
OType     VAL_O    =  VAL
OType  ENABLE_O    =  ENABLE
Type     ENABLE    =  Enable() || Disable()
```

Figure 5-6: TRSpec type definitions for a processor with instruction and data memory interfaces.

```
Rule "Loadi"
     Proc(pc,rf,inst,rport,wport) where Loadi(rd,const)=inst
==>  Proc(pc+1,rf[rd:=const],inst,rport,wport)
Rule "Loadpc"
     Proc(pc,rf,inst,rport,wport) where Loadpc(rd)=inst
==>  Proc(pc+1,rf[rd:=pc],inst,rport,wport)
Rule "Add"
     Proc(pc,rf,inst,rport,wport) where Add(rd,r1,r2)=inst
==>  Proc(pc+1,rf[rd:=(rf[r1]+rf[r2])],inst,rport,wport)
Rule "Sub"
     Proc(pc,rf,inst,rport,wport) where Sub(rd,r1,r2)=inst
==>  Proc(pc+1,rf[rd:=(rf[r1]-rf[r2])],inst,rport,wport)
Rule "Bz-Taken"
     Proc(pc,rf,inst,rport,wport) if (rf[rc]==0)
               where Bz(rc,ra)=inst
==>  Proc(rf[ra],rf,inst,rport,wport)
Rule "Bz-Not-Taken"
     Proc(pc,rf,inst,rport,wport) if (rf[rc]!=0)
               where Bz(rc,ra)=inst
==>  Proc(pc+1,rf,inst,rport,wport)
Rule "Load Start"
     Proc(pc,rf,inst,Rport(-,-,NotBusy( )),wport)
               where Load(rd,ra)=inst
==>  Proc(pc,rf,inst,Rport(rf[ra],-,Busy( )),wport)
Rule "Load Finish"
     Proc(pc,rf,inst,Rport(-,val,Busy( )),wport)
               where Load(rd,ra)=inst
==>  Proc(pc+1,rf[rd:=val],inst,Rport(-,val,NotBusy( )),wport)
Rule "Store Enable"
     Proc(pc,rf,inst,rport,wport) where Store(ra,r)=inst
==>  Proc(pc+1,rf,inst,rport,Wport(rf[ra],rf[r],Enable( )))
Rule "Store Disable"
     Proc(pc,rf,inst,rport,wport) if Store(-,-)!=inst
==>  Proc(pc,rf,inst,rport,Wport(-,-,Disable( )))
```

Figure 5-7: TRSPEC rewrite rules for a processor with instruction and data memory interfaces.

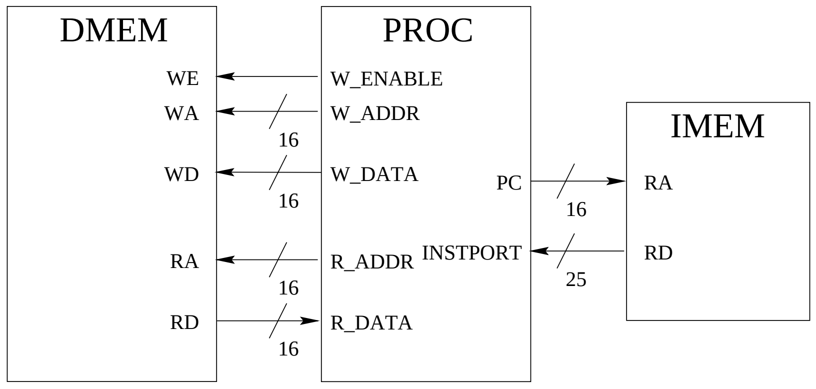


Figure 5-8: A processor's memory interfaces and their connections.

PC_O output port and INSTPORT input port of the synthesized processor should be connected to the external instruction memory module as shown in Figure 5-8. This interface assumes an instruction memory module with combinational lookup.

The new rewrite rule for the execution of the *Add* instruction is

```
Rule "Add"
     Proc(pc, rf, inst, rport, wport) where Add(rd,r1,r2)=inst
==>  Proc(pc+1, rf[rd:=(rf[r1]+rf[r2])], inst, rport, wport)
```

The external instruction memory is indexed by the current value of the PC_O subterm, and the rule looks to the INSTPORT subterm for the instruction returned by the external instruction memory. Besides this difference in instruction lookup, the rule is otherwise identical to the original *Add* rule from Section 2.2.

**Data Memory Interfaces:**

In the new definition of PROC, the internal data memory array has also been replaced by read and write interfaces to an external memory module. The data read-port term, RPORT, is a product term consisting of an output ADDR subterm, an input VAL subterm, and a BUSY status subterm. Their connections to the external memory module are shown in Figure 5-8. To execute a *Load* instruction, the following two rules are needed to manipulate the memory read interface in two steps.

```
Rule "Load Start"
     Proc(pc, rf, inst, Rport(-,-,NotBusy( )), wport)
             where Load(rd,ra)=inst
==>  Proc(pc, rf, inst, Rport(rf[ra],-,Busy( )), wport)

Rule "Load Finish"
     Proc(pc, rf, inst, Rport(-,val,Busy( )), wport)
             where Load(rd,ra)=inst
==>  Proc(pc+1, rf[rd:=val], inst, Rport(-,val,NotBusy( )), wport)
```

| Version | FF (bit) | Util. (%) | Freq. (MHz) |
|---|---|---|---|
| TRSpec | 161 | 60 % | 40.0 |
| Hand-coded RTL | 160 | 50 % | 41.0 |

Figure 5-9: Summary of the processor synthesis results.

When the current instruction is a *Load* instruction, the *Load Start* rule sets the `ADDR` subterm to the load address and sets the `BUSY` status subterm to `Busy( )`. In the second step, the *Load Finish* rule completes the *Load* instruction by updating the register file with the returned load value and by resetting the `BUSY` status term to `NotBusy( )`.

The data write-port term, `WPORT`, is a product term consisting of an output `ADDR` subterm, an output `VAL` subterm, and an output `ENABLE` status term. The following two rules are needed to control this synchronous write interface.

```
Rule "Store Enable"
     Proc(pc, rf, inst, rport, wport) where Store(ra,r)=inst
==>  Proc(pc+1, rf, inst, rport, Wport(rf[ra],rf[r],Enable( )))

Rule "Store Disable"
     Proc(pc, rf, inst, rport, wport) if Store(-,-)!=inst
==>  Proc(pc, rf, inst, rport, Wport(-,-,Disable( )))
```

When the current instruction is a *Store* instruction, the *Store Enable* rule sets the current store address and store data in the write-port term. Furthermore, the *Store Enable* rule enables the write interface by setting the `ENABLE` subterm to `Enable( )`. The *Store Disable* rule resets the `ENABLE` subterm to `Disable( )` after a store operation is completed.

### 5.2.2 Synthesis Results

In this example, the TRSpec framework not only offers the ease of a high-level design flow but also produces a final implementation that is comparable to a hand-crafted effort. A synthesizable Verilog RTL description of the processor can be generated by TRAC from the description in Figures 5-6 and 5-7. The TRAC-generated RTL description is further compiled for implementation in a Xilinx XC4013XL-08 FPGA [Xila] using the Xilinx Foundation 1.5i tools [Xilb]. The row labeled "TRSpec" in Figure 5-9 characterizes the FPGA implementation in terms of the number of flip-flops, the overall resource utilization, and the maximum clock frequency. As a reference, a hand-coded Verilog RTL description of the same processor (included in

Figure 5-10 and 5-11) is also synthesized in this study. The row labeled "Hand-coded RTL" characterizes the implementation synthesized from the hand-coded Verilog description. The data indicate that the TRSpec description results in an FPGA implementation that is similar in size and speed to the result of the hand-coded Verilog description. This similarity should not be surprising because, after all, both descriptions are describing the same processor ISA, albeit under very different design methodologies. In the manual Verilog design flow, a human designer has interpreted the ISA to create an RTL circuit description, but unlike in the simple GCD circuit of the previous section, it is hard for the designer to depart too far from the specification in a design with even modest complexity. Thus, if TRAC can correctly and efficiently interpret the operation-centric ISA description, one should expect TRAC to generate an RTL description that resembles the human designed circuit.

What is not apparent from the circuit characterizations is the difference in development time and effort. The TRSpec and the hand-coded Verilog descriptions are comparable in length. However, the TRSpec description can be translated in a literal fashion from an ISA manual. Whereas, although the hand-coded Verilog description is relatively simple, it has a much weaker correlation to the ISA specification. The hand-coded RTL description also includes circuit implementation information that the RTL designer has to improvise. Whereas, in a TRSpec design flow, the designer can rely on TRAC to supply the implementation-related information. The TRSpec framework permits a natural and intuitive decomposition of hardware behavior into atomic operations with a sequential interpretation. It is TRAC's responsibility to identify operations that can be implemented as concurrent hardware and to insert interlocks between operations that need to be sequentialized. In a traditional design framework, a similar type of analysis and translation must be performed manually by the designer. This not only creates more work for the designer but also creates more opportunity for error.

## 5.3 MIPS R2000 Processor

Appendix B gives the TRSpec description of a five-stage pipelined integer processor core based on the MIPS R2000 ISA (as described in [Kan87]). The description corresponds to the elastic pipeline illustrated in Figure 5-12. The stages of the elastic pipeline are separated by abstract FIFOs, which have a finite but unspecified size. During synthesis, TRAC converts such an asynchronous elastic pipeline description into a synchronous pipeline by instantiating a special FIFO implementation that consists of a single stage of register and flow-control logic. Further information on creating a pipelined description is discussed in Section 6.3. The conversion from an asynchronous pipeline description to a synchronous implementation is described in Section 6.5.1.

```
module PROC (
        WriteData, WriteAddr, WriteEnable,
        ReadAddr, ReadData,
        InstAddr, Inst,
        CLK, _RST
);

output[31:0] WriteData;
output[31:0] WriteAddr;
output       WriteEnable;

output[31:0] ReadAddr;
input[31:0]  ReadData;

output[31:0] InstAddr;
input[22:0]  Inst;

input        CLK;
input        _RST;

reg[31:0]    pc;
reg[31:0]    regFile[0:3];

wire[2:0]    op;
wire[1:0]    rd;
wire[1:0]    r1;
wire[1:0]    r2;
wire[31:0]   rdv;
wire[31:0]   r1v;
wire[31:0]   r2v;
wire[31:0]   immediate;
```

*(Courtesy of D. L. Rosenband.)*

Figure 5-10: Hand-coded Verilog description of a simple processor. (Part 1)

```
assign op=Inst[22:20];
assign rd=Inst[19:18];
assign r1=Inst[3:2];
assign r2=Inst[1:0];
assign immediate={{16{Inst[17]}},Inst[17:2]};
assign rdv=regFile[rd];
assign r1v=regFile[r1];
assign r2v=regFile[r2];
assign      WriteData=r1v;       // store data out;
assign      WriteAddr=rdv;       // store address out;
assign      WriteEnable=(op==6);
assign      ReadAddr=r1v;
assign      InstAddr=pc;

always@(posedge CLK) begin
        case (op)
        0:       regFile[rd]<=immediate;
        1:       regFile[rd]<=pc;
        2:       regFile[rd]<=regFile[r1]+regFile[r2];
        3:       regFile[rd]<=regFile[r1]-regFile[r2];
        5:       regFile[rd]<=ReadData;
        endcase
end

always@(posedge CLK) begin
        if (_RST) begin
                if ((op==4) && (rdv==0)) begin
                        pc<=r1v;
                end else begin
                        pc<=pc+1;
                end
        end else begin
                pc<=0;
        end
end
endmodule
```

*(Courtesy of D. L. Rosenband.)*

Figure 5-11: Hand-coded Verilog description of a simple processor. (Part 2)

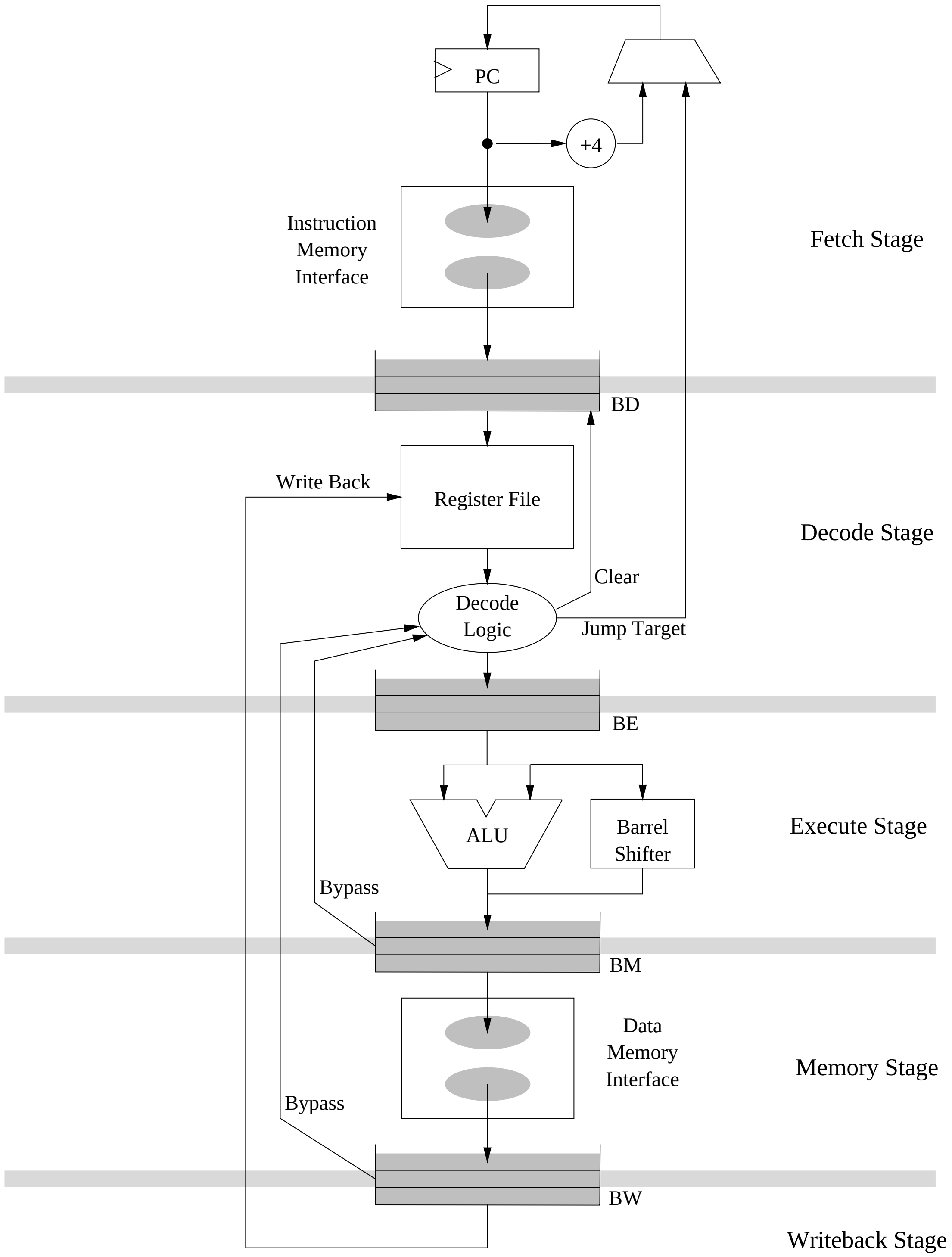


Figure 5-12: Block diagram of the five-stage pipelined MIPS processor core.

### 5.3.1 MIPS Integer Subset

The TRSpec description in Appendix B implements all MIPS R2000 instructions except:

1. Integer multiple and divide instructions (MFHI, MTHI, MFLO, MTLO, MULT, MULTU, DIV, DIVU)
2. Half-word, byte and non-aligned load and store instructions (LB, LH, LWL, LBU, LHU, LWR, SB, SH, SWL, SWR)
3. Privileged instructions (SYSCALL, BREAK)
4. Coprocessor related instructions

The integer core description also does not support exception handling, privileged mode or memory management. The semantics of the memory load and branch/jump instructions has been altered to eliminate delay slots. In other words, the result of a load instruction is immediately available to the next instruction, and the effect of a branch/jump instruction takes place immediately.

### 5.3.2 Microarchitecture

The description corresponds to an implementation of the MIPS ISA in a five-stage pipelined Harvard microarchitecture. The description specifies separate external instruction and data memory interfaces that are similar to the scheme in Section 5.2. The rewrite rules imply a register file usage that requires two combinational read ports and one synchronous write port.

**Fetch Stage:** A single rule describes the sequential instruction fetch using the instruction fetch interface.

**Decode Stage:** Separate rules specify the decoding of the different instruction subclasses. When a read-after-write hazard is detected, the decode-stage rules attempt to bypass completed data from the execute, memory and write-back stages. If read-after-write hazards cannot be resolved by bypassing, the decode-stage rules stop firing. Branch or jump instructions are also carried out by decode-stage rules. After a control flow instruction, one bubble is inserted into the pipeline before the execution can restart at the correct jump target.

**Execute Stage:** Execute-stage rules describe the execution of various ALU instructions. Separate execute-stage rules also describe memory address calculations for load and store instructions. The type definition of the MIPS processor term includes a user-defined abstract type `SHIFTER`. The `SHIFTER` abstract type encapsulates a barrel shifter implemented in Verilog. The execute-stage rules access the `SHIFTER` term's interface to compute arithmetic and logical shifts of integer operands.

|  | CBA tc6a |  | LSI 10K |  |
|---|---|---|---|---|
|  | area | speed | area | speed |
| version | (cell) | (MHz) | (cell) | (MHz) |
| TRSpec | 9059 | 96.6 | 34674 | 41.9 |
| Hand-coded RTL | 7168 | 96.0 | 26543 | 42.1 |

Figure 5-13: Summary of MIPS synthesis results.

**Memory Stage:** Load and store instructions are executed. Other instructions simply pass through this stage.

**Write-Back Stage:** All register updates are performed in the write-back stage.

### 5.3.3 Synthesis Results

The TRSpec description of the MIPS core can be compiled by TRAC into a synthesizable Verilog RTL description. The synthesizable Verilog description can then be compiled by the *Synopsys Design Compiler* [Synb] to target both the *Synopsys CBA* [Syna] and *LSI Logic 10K* [LSI] gate-array libraries. For comparison, a hand-coded Verilog RTL description of the same MIPS microarchitecture is also compiled for the same technology libraries. Figure 5-13 summarizes the pre-layout area and speed estimates reported by the *Synopsys Design Compiler*. The row labeled "TRSpec" characterizes the implementation synthesizes from the TRSpec description. The row labeled "Hand-coded RTL" characterizes the implementation synthesized from the hand-coded Verilog description.

As is the case for the simple processor in the previous section, the results from synthesizing the TRSpec description and the hand-coded Verilog description are in good agreement, especially in terms of cycle time.[2] The implementation synthesized from the hand-coded Verilog RTL description is 20 to 25 percent smaller than the implementation synthesized from the TRSpec description. The TRSpec and the hand-coded Verilog descriptions are similar in length (790 vs. 930 lines of source code), but the TRSpec description is developed in less than one day (eight hours), whereas the hand-coded Verilog description requires nearly five days to complete.

[2]Both Synopsys synthesis runs are configured for *high-effort* on minimizing cycle time.

## 5.4 Summary

This chapter presents the results from applying TRAC to synthesize TRSpec descriptions. The designs are targeted for implementation technologies that include Xilinx FPGAs and ASIC gate-array libraries. The quality of TRAC-generated implementations is evaluated against reference implementations synthesized from hand-coded Verilog RTL descriptions.

As part of this study, several examples have also been targeted for the Wildcard Reconfigurable Computing Engine from Annapolis Micro Systems [Ann]. The Wildcard hardware contains a single Xilinx Vertex XCV300-4 FPGA packaged in a PCMCIA form-factor. (Higher capacity devices are available on PCI cards.) The Wildcard hardware can be plugged into standard expansion slots of personal computers, and FPGA configurations can be created and uploaded onto the Wildcard FPGA from the main processor. The FPGA configuration can include memory-mapped I/O and DMA functionalities so a software application on the main processor can interface with the hardware application on the FPGA interactively. Such a flexible reconfigurable hardware platform perfectly complements the ability to rapidly create hardware designs in the TRSpec framework.

In one scenario, algorithmic descriptions in TRSpec, like Euclid's Algorithm from Section 5.1, can be synthesized for the Wildcard FPGA. This effectively creates a hardware-software co-processing environment where an application running on the processor can launch hardware-assisted computations on the FPGA hardware. In this usage, TRSpec provides the means for an application developer to retarget suitable parts of an application for hardware acceleration, expending only comparable time and effort as software development.

In another usage, an architect can create simulatable and synthesizable hardware prototypes from architectural descriptions in TRSpec. For example, the TRSpec description of the MIPS processor from Section 5.3 can be synthesized for execution on the Wildcard FPGA. In this context, the Wildcard FPGA becomes a hardware emulator where actual MIPS binaries can be executed. New mechanisms and ideas can be quickly added to the FPGA-emulated prototype by making high-level modifications to the architectural-level TRSpec description.

# Chapter 6

# Microprocessor Design Exploration

This chapter demonstrates the application of operation-centric hardware description and synthesis in microprocessor design. A high-level TRSPEC description of an instruction set architecture (ISA) is amenable to transformations that produce descriptions of pipelined and superscalar processors. This ability to rapidly create derivative designs enables a feedback-driven iterative approach to custom microprocessor development.

## 6.1 Design Flow Overview

In this design flow, an architect starts by formulating a high-level ISA specification as a TRS. The goal at this stage is to define an ISA precisely without injecting implementation details. For example, the rewrite rules from Section 2.2 and the type definitions from Example 4.3, together, constitute an ISA specification in the TRSPEC language. Interpreting the specification as is, TRAC synthesizes the register-transfer level (RTL) implementation of a single-issue, non-pipelined processor.

Based on this ISA description, the architect can further derive TRSPEC descriptions of pipelined processors by introducing pipeline buffers (as described in Sections 2.3 and 2.4). The TRSPEC framework simplifies the insertion of pipeline stages by allowing the architect to create *elastic* pipelines where pipeline stages are separated by FIFOs. The operations in one pipeline stage can be described independently of the operations in the other stages. During synthesis, TRAC maps an asynchronous elastic pipeline description onto a synchronous pipeline where the stages are separated by simple pipeline registers.

A pipelined processor description in TRSPEC can be further transformed into a superscalar description by adding new rules derived from composing existing rules from the same pipeline stage. A composite rule, when applied, has the same effect as the sequential in-order execution of its constituent rules. The predicate of a composite rule is only enabled in a state where the full sequence of rules can be applied. Thus, the correctness of the expanded description is guaranteed because adding composite rules cannot introduce any new behavior.

Both pipelining and superscalar design derivations are performed as source-to-

```
Type   PROC  =  Proc(PC, RF, IMEM, DMEM)
Type     PC  =  Bit[32]
Type     RF  =  Array [RNAME] VAL
Type  RNAME  =  Reg0( ) || Reg1( ) || Reg2( ) || Reg3( )
Type    VAL  =  Bit[32]
Type   IMEM  =  Array [PC] INST
Type   INST  =  Loadi(RNAME,VAL)
             || Loadpc(RNAME)
             || Add(RNAME,RNAME,RNAME)
             || Sub(RNAME,RNAME,RNAME)
             || Bz(RNAME,RNAME)
             || Load(RNAME,RNAME)
             || Store(RNAME,RNAME)
Type   DMEM  =  Array [ADDR] VAL
Type   ADDR  =  Bit[32]
```

Figure 6-1: Type definitions for a simple non-pipelined processor.

source transformations in the TRSpec language. The derived designs can be compiled into Verilog RTL descriptions using TRAC. For design feedback, the generated Verilog descriptions can be simulated and evaluated using commercial tools like the *Cadence Affirma NC Verilog Simulator* [Cad] and the *Synopsys RTL Analyzer* [Sync].

## 6.2 Step 1: ISA Specification

Figures 6-1 and 6-2 repeat the rules and type definitions of a simple ISA, already presented in Section 2.2 and Example 4.3. The type definitions in Figure 6-1 have been altered to increase the processor data width from 16 to 32 bits. For conciseness, the rules in Figure 6-2 are given in an abbreviated format where all rules share a common left-hand-side pattern, given once at the top. When synthesized, the TRSpec description roughly corresponds to the datapath shown in Figure 6-3.

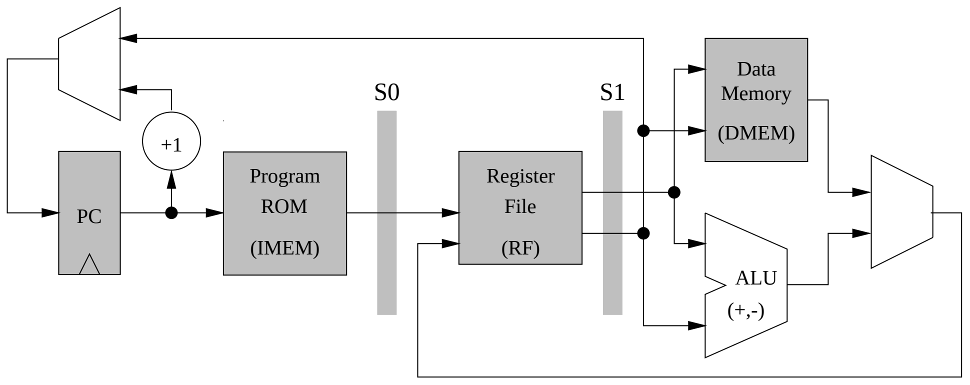
## 6.3 Step 2: Pipelining Transformation

The TRSpec processor description from the previous section can be pipelined by splitting each rule into multiple sub-rules where each sub-rule describes the sub-operation in a pipeline stage. As in Section 2.4, the processing of an instruction can be broken down into separate fetch and execute sub-operations in a two-stage pipelined design. The pipelined design needs buffers to hold partially executed instructions. In a TRSpec description, the pipeline buffers are modeled as FIFOs of a finite

Proc(pc, rf, imem, dmem)

| | |
|---|---|
| *Loadi:* | where Loadi(rd,const) = imem[pc] |
| | → Proc(pc+1, rf[rd:=const], imem, dmem) |
| *Loadpc:* | where Loadpc(rd) = imem[pc] |
| | → Proc(pc+1, rf[rd:=pc], imem, dmem) |
| *Add:* | where Add(rd,r1,r2) = imem[pc] |
| | → Proc(pc+1, rf[rd:=rf[r1]+rf[r2]], imem, dmem) |
| *Sub:* | where Sub(rd,r1,r2) = imem[pc] |
| | → Proc(pc+1, rf[rd:=rf[r1]-rf[r2]], imem, dmem) |
| *Bz-Taken:* | if rf[rc]=0 where Bz(rc,rt) = imem[pc] |
| | → Proc(rf[rt], rf, imem, dmem) |
| *Bz-Not-Taken:* | if rf[rc]≠0 where Bz(rc,rt) = imem[pc] |
| | → Proc(pc+1, rf, imem, dmem) |
| *Load:* | where Load(rd,ra) = imem[pc] |
| | → Proc(pc+1, rf[rd:=dmem[rf[ra]]], imem, dmem) |
| *Store:* | where Store(ra,r) = imem[pc] |
| | → Proc(pc+1, rf, imem, dmem[rf[ra]:=rf[r]]) |

Figure 6-2: Rules for a simple non-pipelined processor.



*(S0 and S1 are potential sites for pipeline buffers.)*

Figure 6-3: A simple processor datapath shown without its control paths.

```
Type    PROC₂ =  Proc₂(PC, RF, BF, IMEM, DMEM)
Type       BF =  Fifo ITEMP
Type    ITEMP =  TLoadi(RNAME,VAL)
              || TAdd(RNAME,VAL,VAL)
              || TSub(RNAME,VAL,VAL)
              || TBz(VAL,PC)
              || TLoad(RNAME,ADDR)
              || TStore(ADDR,VAL)
```

Figure 6-4: Additional type definitions for the two-stage pipelined processor.

but unspecified size. In the synthesis phase, TRAC replaces these FIFOs by simple pipeline registers and flow control logic. In the description phase, the FIFO-based elastic pipeline abstraction allows the operations in different stages to be described independently without references to the operations in the other stages. A rule that describes an operation in a particular pipeline stage typically dequeues from the up-stream FIFO and enqueues into the down-stream FIFO.

To describe a two-stage Fetch/Execute pipeline, the type of the processor term is redefined as $\mathsf{PROC}_2$ in Figure 6-4. In contrast to PROC in Figure 6-1, a $\mathsf{PROC}_2$-typed term contains an additional BF-typed field. BF is a FIFO that holds decoded instruction templates whose operands have been fetched from the register file. As discussed in Section 2.4, the original *Add* rule from the ISA specification may be replaced by the following two rules, corresponding to the fetch and execute sub-operations, respectively:

*Add Fetch:*

$$\begin{array}{ll} & \mathsf{Proc_2(pc, rf, bf, imem, dmem)} \\ & \qquad\qquad \mathsf{if} \quad \mathsf{r1} \notin \mathit{Target}(\mathsf{bf}) \wedge \mathsf{r2} \notin \mathit{Target}(\mathsf{bf}) \\ & \qquad\qquad \mathsf{where} \quad \mathsf{Add(rd,r1,r2)} = \mathsf{imem[pc]} \\ \rightarrow & \mathsf{Proc_2(pc{+}1, rf, bf;TAdd(rd,rf[r1],rf[r2]), imem, dmem)} \end{array}$$

*Add Execute:*

$$\begin{array}{ll} & \mathsf{Proc_2(pc, rf, TAdd(rd,v1,v2);bf, imem, dmem)} \\ \rightarrow & \mathsf{Proc_2(pc, rf[rd{:=}v1{+}v2], bf, imem, dmem)} \end{array}$$

Splitting the effect of one rewrite rule into multiple rules destroys the atomicity of the original rule and thus can cause new behaviors that may not conform to the original specification. Therefore, in addition to determining the appropriate division of work among the pipeline stages, the architect must also resolve any newly created hazards. For example, the fetch rule's predicate expression has been extended to check if the source register names, r1 and r2, are in *Target*(bf). (*Target*(bf) is the shorthand for the set of target register names in bf.) This extra predicate condition stalls instruction fetching when a RAW (read-after-write) hazard exists.

The *Bz-Taken* rule and the *Bz-Not-Taken* rule in Figure 6-2 can also be split into their fetch and execute sub-operations. Both *Bz* rules share the following instruction fetch rule:

*Bz Fetch:*

$$\begin{array}{lrcl}
 & \multicolumn{3}{l}{\mathsf{Proc_2(pc, rf, bf, imem, dmem)}} \\
 & \mathsf{if} & \multicolumn{2}{l}{\mathsf{rc} \notin \mathit{Target}(\mathsf{bf}) \wedge \mathsf{rt} \notin \mathit{Target}(\mathsf{bf})} \\
 & \mathsf{where} & \mathsf{Bz(rc,rt)} = & \mathsf{imem[pc]} \\
\rightarrow & \multicolumn{3}{l}{\mathsf{Proc_2(pc{+}1, rf, bf;TBz(rf[rc],rf[rt]), imem, dmem)}}
\end{array}$$

The two execute rules for the *Bz* instruction are

*Bz-Taken Execute:*

$$\begin{array}{lrl}
 & \multicolumn{2}{l}{\mathsf{Proc_2(pc, rf, TBz(vc,vt);bf, imem, dmem)}} \\
 & \mathsf{if} & \mathsf{vc} = 0 \\
\rightarrow & \multicolumn{2}{l}{\mathsf{Proc_2(vt, rf, \epsilon, imem, dmem)}}
\end{array}$$

and

*Bz-Not-Taken Execute:*

$$\begin{array}{lrl}
 & \multicolumn{2}{l}{\mathsf{Proc_2(pc, rf, TBz(vc,vt);bf, imem, dmem)}} \\
 & \mathsf{if} & \mathsf{vc} \neq 0 \\
\rightarrow & \multicolumn{2}{l}{\mathsf{Proc_2(pc, rf, bf, imem, dmem)}}
\end{array}$$

All of the rules in Figure 6-2 can be partitioned into separate fetch and execute sub-rules to completely convey the operations of a two-stage pipelined processor. The current partitioning places the pipeline buffer (bf) at the position labeled *S1* in Figure 6-3. Pipelines with different number of stages and buffer placements can also be derived similarly.

A generic instruction fetch rule is

$$\begin{array}{lrcl}
 & \multicolumn{3}{l}{\mathsf{Proc_2(pc, rf, bf, imem, dmem)}} \\
 & \mathsf{if} & \multicolumn{2}{l}{(\mathit{Source}(\mathsf{inst}) \not\cap \mathit{Target}(\mathsf{bf})\ )} \\
 & \mathsf{where} & \mathsf{inst} = & \mathsf{imem[pc]} \\
\rightarrow & \multicolumn{3}{l}{\mathsf{Proc_2(pc{+}1, rf, bf;}\mathit{Decode}\mathsf{(inst), imem, dmem)}}
\end{array}$$

*Source*(inst) is the shorthand to extract the source register names from instruction inst. *Decode*(inst) is the shorthand that maps inst to its corresponding instruction template where the register operands have been fetched. For example, the expression '*Decode*(Add(rd,r1,r2))' is the same as 'TAdd(rd,rf[r1],rf[r2])'. The execute-stage sub-rules for all instructions are given in Figure 6-5.

## 6.4 Step 3: Superscalar Transformation

This section describes the transformation from a pipelined microarchitecture to a pipelined superscalar microarchitecture. The transformation produces a microarchi-

```
Proc2(pc, rf, bf, imem, dmem) where itemp;rest = bf

Loadi:          where TLoadi(rd,v) = itemp
                → Proc2(pc, rf[rd:=v], rest, imem, dmem)
Add:            where TAdd(rd,v1,v2) = itemp
                → Proc2(pc, rf[rd:=v1+v2], rest, imem, dmem)
Sub:            where TSub(rd,v1,v2) = itemp
                → Proc2(pc, rf[rd:=v1-v2], rest, imem, dmem)
Bz-Taken:       if vc=0  where TBz(vc,vt) = itemp
                → Proc2(vt, rf, ε, imem, dmem)
Bz-Not-Taken:   if vc≠0  where TBz(vc,vt) = itemp
                → Proc2(pc, rf, rest, imem, dmem)
Load:           where TLoad(rd,va) = itemp
                → Proc2(pc, rf[rd:=dmem[va]], rest, imem, dmem)
Store:          where TStore(va,v) = itemp
                → Proc2(pc, rf, rest, imem, dmem[va:=v])
```

Figure 6-5: Rules for the execute stage of the two-stage pipelined processor.

tecture similar to the DEC Alpha 21064 [DWA+92] in that the microarchitecture processes multiple instructions in each pipeline stage when possible, but does not allow out-of-order execution. To derive a two-way superscalar processor description from a pipelined processor description, one needs to compose two rules from the same pipeline stage into a new composite rule that combines the effects of both rules. Given that TRAC generates RTL descriptions where the entire effect of a rule is executed in one clock cycle, the composite rule yields an RTL design that is capable of processing two instructions per clock cycle.

### 6.4.1 Derivation of Composite Rules

A TRS rule $r$ on a set of terms $\mathcal{T}$ can be described by a function $f$ whose domain $D$ and image $I$ are subsets of $\mathcal{T}$. Given a rule:

$$s \text{ } \mathit{if} \text{ } p \;\rightarrow\; s'$$

the function $f$ may be expressed as

$$f(s) = \mathit{if}\ \pi(s)\ \mathit{then}\ \delta(s)\ \mathit{else}\ s$$

where $\pi(\,)$ represents the firing condition derived from the left-hand-side pattern $s$ and the predicate expression $p$; and $\delta(\,)$ represents the function that computes the new term. Given two rules $r_1$ and $r_2$, the composite rule $r_{1,2}$ can be described by the function $f_{1,2}$ where

$$
\begin{aligned}
f_{1,2}(s) = \;& \textit{if}\ \pi_1(s)\ \textit{then} \\
& \quad\quad \textit{if}\ \pi_2(\delta_1(s))\ \textit{then} \\
& \quad\quad\quad \delta_2(\delta_1(s)) \\
& \quad\quad \textit{else} \\
& \quad\quad\quad s \\
& \textit{else} \\
& \quad s \\
= \;& \textit{if}\ \pi_1(s) \wedge \pi_2(\delta_1(s))\ \textit{then} \\
& \quad \delta_2(\delta_1(s)) \\
& \textit{else} \\
& \quad s
\end{aligned}
$$

Let $D_1$ and $I_1$ be the domain and image of $f_1$ and $D_2$ and $I_2$ be the domain and image of $f_2$, the domain $D_{1,2}$ of $f_{1,2}$ is the subset of $D_1$ that produces the restricted image $I_1 \cap D_2$ using $f_1$. By this definition of composition, adding $r_{1,2}$ to a TRS that already contains $r_1$ and $r_2$ does not introduce any new behaviors since all transitions admitted by $r_{1,2}$ can be simulated by consecutive applications of $r_1$ and $r_2$. However, $r_1$ and $r_2$ cannot be replaced by $r_{1,2}$ because some behaviors could be eliminated. Removing $r_{1,2}$ may create a deadlock or a livelock.

Rule composition can also be described as a purely syntactic operation. Given the following two rewrite rules:

$$
\begin{array}{ll}
s_1\ \textit{if}\ p_1\ \rightarrow\ s_1' & (r_1) \\
s_2\ \textit{if}\ p_2\ \rightarrow\ s_2' & (r_2)
\end{array}
$$

one first derives a restricted instance of $r_2$ that is directly applicable to $s_1'$ such that

$$s_1'\ \textit{if}\ p_2'\ \rightarrow\ s_2'' \qquad (\textit{restricted instance of}\ r_2)$$

This instance of $r_2$ can then be composed with $r_1$ as follows:

$$s_1\ \textit{if}\ (p_1 \wedge p_2')\ \rightarrow\ s_2'' \qquad (r_{1,2})$$

### 6.4.2 A Composite Rule Example

Consider the *Add Fetch* and *Bz Fetch* rules of the two-stage pipelined processor from Section 6.3.

```
Add Fetch:
    Proc₂(pc, rf, bf, imem, dmem)
                        if    r1∉Target(bf) ∧ r2∉Target(bf)
                     where    Add(rd,r1,r2)  =  imem[pc]
 →  Proc₂(pc+1, rf, bf;TAdd(rd,rf[r1],rf[r2]), imem, dmem)
```

```
Bz Fetch:
    Proc₂(pc, rf, bf, imem, dmem)
                        if    rc∉Target(bf) ∧ rt∉Target(bf)
                     where    Bz(rc,rt)  =  imem[pc]
 →  Proc₂(pc+1, rf, bf;TBz(rf[rc],rf[rt]), imem, dmem)
```

One can rewrite the *Bz Fetch* rule as if it is being applied to the right-hand-side expression of the *Add Fetch* rule. The restricted *Bz Fetch* rule appears as

```
    Proc₂(pc+1, rf, bf;TAdd(rd,rf[r1],rf[r2]), imem, dmem)
                        if    rc∉Target(bf;TAdd(rd,rf[r1],rf[r2]))
                        ∧     rt∉Target(bf;TAdd(rd,rf[r1],rf[r2]))
                     where    Bz(rc,rt)  =  imem[pc+1]
 →  Proc₂((pc+1)+1, rf,
           (bf;TAdd(rd,rf[r1],rf[r2]));TBz(rf[rc],rf[rt]),
           imem, dmem)
```

This rule is more specific than the original *Bz Fetch* rule because bf is required to contain an *Add* instruction template as the youngest entry. A more specific instance of a TRS rule is guaranteed to be correct because it fires under fewer conditions. The *Add Fetch* and *Bz Fetch* rules can be combined into a composite rule:

```
    Proc₂(pc, rf, bf, imem, dmem)
                        if    r1∉Target(bf)  ∧  r2∉Target(bf)
                        ∧     rc∉Target(bf;TAdd(rd,rf[r1],rf[r2]))
                        ∧     rt∉Target(bf;TAdd(rd,rf[r1],rf[r2]))
                     where    Add(rd,r1,r2)  =  imem[pc]
                                  Bz(rc,rt)  =  imem[pc+1]
 →  Proc₂((pc+1)+1, rf,
           (bf;TAdd(rd,rf[r1],rf[r2]));TBz(rf[rc],rf[rt]),
           imem, dmem)
```

The predicate expression in the rule above can be simplified as shown in the rule below by interpreting the effect of enqueuing to an abstract FIFO term.

$$
\begin{array}{lrl}
& \multicolumn{2}{l}{\mathsf{Proc_2(pc, rf, bf, imem, dmem)}} \\
& \textsf{if} & \mathsf{r1} \notin \mathit{Target}(\mathsf{bf}) \ \wedge\ \mathsf{r2} \notin \mathit{Target}(\mathsf{bf}) \\
& \wedge & \mathsf{rt} \notin \mathit{Target}(\mathsf{bf}) \ \wedge\ \mathsf{rc} \notin \mathit{Target}(\mathsf{bf}) \\
& \wedge & \mathsf{rc} \neq \mathsf{rd} \ \wedge\ \mathsf{rt} \neq \mathsf{rd} \\
& \textsf{where} & \mathsf{Add(rd,r1,r2)} = \mathsf{imem[pc]} \\
& & \mathsf{Bz(rc,rt)} = \mathsf{imem[pc+1]} \\
\rightarrow & \multicolumn{2}{l}{\mathsf{Proc_2((pc+1)+1, rf,}} \\
& \multicolumn{2}{l}{\quad \mathsf{(bf;TAdd(rd,rf[r1],rf[r2]));TBz(rf[rc],rf[rt]),}} \\
& \multicolumn{2}{l}{\quad \mathsf{imem, dmem)}}
\end{array}
$$

In an implementation synthesized according to the procedures outlined in Chapter 3, the scheduler should give higher priority to a composite rule over its constituent rules when they are enabled in the same clock cycle.

### 6.4.3 Derivation of a Two-Way Superscalar Processor

This section presents the derivation of the composite rules for a two-way superscalar processor description. The derivations are based on the two-stage pipelined processor from Section 6.3. For each of the two pipeline stages, different combinations of two rules from the same stage are composed. In general, given a pipeline stage with $N$ rules, a superscalar transformation leads to an $O(N^s)$ increase in the number of rules where $s$ is the degree of superscalarity. Since superscalar transformation implies an increase in hardware resources like register file ports, ALUs and memory ports, one may not want to compose all possible combinations of rules. For example, one may not want to compose a memory load rule with another memory load rule if the memory interface can only accept one operation per cycle.

This derivation assumes that there are no restrictions on hardware resources except that the data memory can only service one operation, a read or a write, in each clock cycle. The derivation also assumes the instruction memory can return two consecutive instruction words on any address alignment.

The generic instruction fetch rule from the end of Section 6.3 can be composed with itself to produce a two-way superscalar fetch rule:

$$
\begin{array}{lrl}
& \multicolumn{2}{l}{\mathsf{Proc_2(pc, rf, bf, imem, dmem)}} \\
& \textsf{if} & \mathit{Source}(\mathsf{inst}) \not\cap \mathit{Target}(\mathsf{bf}) \\
& \wedge & \mathit{Source}(\mathsf{inst'}) \not\cap (\mathit{Target}(\mathsf{bf}) \cup \mathit{Target}(\mathsf{inst})) \\
& \textsf{where} & \mathsf{inst} = \mathsf{imem[pc]} \\
& & \mathsf{inst'} = \mathsf{imem[pc+1]} \\
\rightarrow & \multicolumn{2}{l}{\mathsf{Proc_2((pc+1)+1, rf,}} \\
& \multicolumn{2}{l}{\quad \mathsf{bf;}\mathit{Decode}(\mathsf{inst});\mathit{Decode}(\mathsf{inst'}), \mathsf{imem, dmem)}}
\end{array}
$$

The superscalar execute rules are derived by composing all legal combinations of the rules in Figure 6-5. A composite execute rule examines both the first and second

| Proc$_2$(pc, rf, bf, imem, dmem) where TAdd(rd,v1,v2);itemp;rest = bf | |
|---|---|
| *Loadi:* | where TLoadi(rd',v) = itemp<br>$\rightarrow$ Proc$_2$(pc, rf[rd:=(v1+v2),rd':=v)], rest, imem, dmem) |
| *Add:* | where TAdd(rd',v1',v2') = itemp<br>$\rightarrow$ Proc$_2$(pc, rf[rd:=(v1+v2),rd':=(v1'+v2')], rest, imem, dmem) |
| *Sub:* | where TSub(rd',v1',v2') = itemp<br>$\rightarrow$ Proc$_2$(pc, rf[rd:=(v1+v2),rd':=(v1'-v2')], rest, imem, dmem) |
| *Bz-Taken:* | if vc=0 where TBz(vc,vt) = itemp<br>$\rightarrow$ Proc$_2$(vt, rf[rd:=v1+v2], $\epsilon$, imem, dmem) |
| *Bz-Not-Taken:* | if vc≠0 where TBz(vc,vt) = itemp<br>$\rightarrow$ Proc$_2$(pc, rf[rd:=v1+v2], rest, imem, dmem) |
| *Load:* | where TLoad(rd',va) = itemp<br>$\rightarrow$ Proc$_2$(pc, rf[rd:=(v1+v2),rd':=dmem[va]], rest, imem, dmem) |
| *Store:* | where TStore(va,v) = itemp<br>$\rightarrow$ Proc$_2$(pc, rf[rd:=v1+v2], rest, imem, dmem[va:=v]) |

Figure 6-6: Combining the *Add Execute* rule with other execute rules.

instruction templates in the pipeline buffer bf. If the first and second instruction templates satisfy the rule's predicate expression, the rule is applied to process both instruction templates simultaneously.

The table in Figure 6-6 gives the composition of the *Add Execute* rule with other execute rules. (Similar composite rules can be derived for the *Loadi Execute* or *Sub Execute* rules.) If the first instruction template in bf is an *Add* instruction template then the second instruction template, when present, can always be executed concurrently. In the composite rules, the expression a[i:=v,i':=v'] denotes a sequential update of location i and i' of array a. If i is the same as i' then a[i:=v,i':=v'] has the same effect as a[i':=v'].

The *Bz-Taken Execute* rule cannot be composed with any other execute rule. If the first position of bf contains the instruction template of a taken branch, bf will subsequently be cleared by the *Bz-Taken Execute* rule. Since every execute-stage rule requires the pipeline buffer to be not empty, none of the execute-stage rules can be applicable immediately after the *Bz-Taken Execute* rule has been applied.

Executing the *Bz-Not-Taken Execute* rule produces no side-effects other than removing the current *Bz* instruction template from the head of bf. Hence, as shown in Figure 6-7, composing a *Bz-Not-Taken Execute* rule with any other rule results in a composite rule that is nearly identical to the second rule in the composition.

The tables in Figures 6-8 and 6-9 give the composition of the *Load Execute* and the *Store Execute* rules with other execute rules. Since the data memory only responds to one memory operation per clock cycle, one cannot compose the *Load Execute* rule or the *Store Execute* rule with another memory access rule.

| Proc$_2$(pc, rf, bf, imem, dmem) where TBz(vc,vt);itemp;rest = bf | |
|---|---|
| *Loadi:* | if vc$\neq$0 where TLoadi(rd,v) = itemp |
| | $\rightarrow$ Proc$_2$(pc, rf[rd:=v], rest, imem, dmem) |
| *Add:* | if vc$\neq$0 where TAdd(rd,v1,v2) = itemp |
| | $\rightarrow$ Proc$_2$(pc, rf[rd:=v1+v2], rest, imem, dmem) |
| *Sub:* | if vc$\neq$0 where TSub(rd,v1,v2) = itemp |
| | $\rightarrow$ Proc$_2$(pc, rf[rd:=v1-v2], rest, imem, dmem) |
| *Bz-Taken:* | if vc$\neq$0 $\wedge$ vc'=0 where TBz(vc',vt') = itemp |
| | $\rightarrow$ Proc$_2$(vt', rf, $\epsilon$, imem, dmem) |
| *Bz-Not-Taken:* | if vc$\neq$0 $\wedge$ vc'$\neq$0 where TBz(vc',vt') = itemp |
| | $\rightarrow$ Proc$_2$(pc, rf, rest, imem, dmem) |
| *Load:* | if vc$\neq$0 where TLoad(rd,va) = itemp |
| | $\rightarrow$ Proc$_2$(pc, rf[rd:=dmem[va]], rest, imem, dmem) |
| *Store:* | if vc$\neq$0 where TStore(va,v) = itemp |
| | $\rightarrow$ Proc$_2$(pc, rf, rest, imem, dmem[va:=v]) |

Figure 6-7: Combining the *Bz-Not-Taken Execute* rule with other execute rules.

| Proc$_2$(pc, rf, bf, imem, dmem) where TLoad(rd,va);itemp;rest = bf | |
|---|---|
| *Loadi:* | where TLoadi(rd',v) = itemp |
| | $\rightarrow$ Proc$_2$(pc, rf[rd:=dmem[va],rd':=v], rest, imem, dmem) |
| *Add:* | where TAdd(rd',v1,v2) = itemp |
| | $\rightarrow$ Proc$_2$(pc, rf[rd:=dmem[va],rd':=(v1+v2)], rest, imem, dmem) |
| *Sub:* | where TSub(rd',v1,v2) = itemp |
| | $\rightarrow$ Proc$_2$(pc, rf[rd:=dmem[va],rd':=(v1-v2)], rest, imem, dmem) |
| *Bz-Taken:* | if vc=0 where TBz(vc,vt) = itemp |
| | $\rightarrow$ Proc$_2$(vt, (rf[rd:=dmem[va]]), $\epsilon$, imem, dmem) |
| *Bz-Not-Taken:* | if vc$\neq$0 where TBz(vc,vt) = itemp |
| | $\rightarrow$ Proc$_2$(pc, (rf[rd:=dmem[va]]), rest, imem, dmem) |

Figure 6-8: Combining the *Load Execute* rule with other execute rules.

| Proc$_2$(pc, rf, bf, imem, dmem) where TStore(va,v);itemp;rest = bf | |
|---|---|
| *Loadi:* | where TLoadi(rd,v') = itemp |
| | $\rightarrow$ Proc$_2$(pc, rf[rd:=v'], rest, imem, dmem[va:=v]) |
| *Add:* | where TAdd(rd,v1,v2) = itemp |
| | $\rightarrow$ Proc$_2$(pc, rf[rd:=v1+v2], rest, imem, dmem[va:=v]) |
| *Sub:* | where TSub(rd,v1,v2) = itemp |
| | $\rightarrow$ Proc$_2$(pc, rf[rd:=v1-v2], rest, imem, dmem[va:=v]) |
| *Bz-Taken:* | if vc=0  where TBz(vc,vt) = itemp |
| | $\rightarrow$ Proc$_2$(vt, rf, $\epsilon$, imem, dmem[va:=v]) |
| *Bz-Not-Taken:* | if vc$\neq$0  where TBz(vc,vt) = itemp |
| | $\rightarrow$ Proc$_2$(pc, rf, rest, imem, dmem[va:=v]) |

Figure 6-9: Combining the *Store Execute* rule with other execute rules.

## 6.5 Synthesis and Analysis

The TRSpec processor descriptions presented in this chapter can be compiled into synthesizable Verilog RTL descriptions using TRAC. The TRAC-generated RTL descriptions can be further compiled, by commercial hardware compilers, for a number of target technologies ranging from ASICs to FPGAs. In addition, the RTL descriptions can also be targeted for the *Synopsys GTECH Library*, a technology-independent logic representation. The GTECH netlist of a design can then be processed by the *Synopsys RTL Analyzer* [Sync] to provide quantitative feedback about the circuit's size and delay.

### 6.5.1 TRSpec to RTL

**Input and Output:**

As in the example from Section 5.2, realistically, the architectural-level descriptions in this chapter need to be modified to access external memory modules through input/output ports. For the single-issue pipelined and non-pipelined processor descriptions, modifications to include instruction and data memory interfaces are similar to the presentation in Section 5.2. On the other hand, for a two-way superscalar processor description, the modification needs to provide an instruction fetch interface that can return two instructions at a time. It is sufficient to have a single fetch interface that can return two consecutive instructions on any address alignment because the two-way superscalar fetch rule only references consecutive locations, pc and pc+1.

**Synchronous Pipeline Synthesis:**

In pipelined processor descriptions, the operation of the processors cannot depend on the exact depth of the pipeline FIFOs. This allows TRAC to instantiate one-deep FIFOs (i.e., a single register) as pipeline buffers. Flow control logic is added to

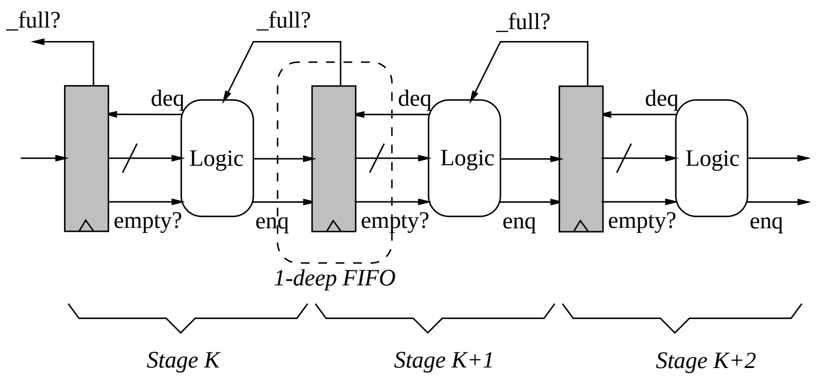


Figure 6-10: Synchronous pipeline with local feedback flow control.

ensure a FIFO is not overflowed or underflowed by enqueue and dequeue operations. A straightforward attempt might lead to the circuit shown in Figure 6-10. In this naive mapping, the one-deep FIFO is full if its register holds valid data; the FIFO is empty if its register holds a *bubble*. With only local flow control between neighboring stages, the overall pipeline would contain a bubble in every other stage during steady-state execution. For example, if pipeline buffer $K$ and $K+1$ are occupied and buffer $K+2$ is empty in some clock cycle, the operation in stage $K+1$ would be enabled to advance at the clock edge, but the operation in stage $K$ is held back because buffer $K+1$ appears full during the clock cycle. The operation in stage $K$ is not enabled until the next clock cycle when buffer $K+1$ has been emptied.

It is important that TRAC creates a flow control logic that includes a combinational multi-stage feedback path that propagates from the last pipeline stage to the first pipeline stage. The cascaded feedback scheme shown in Figure 6-11 allows stage $K$ to advance both when pipeline buffer $K+1$ is actually empty and when buffer $K+1$ is going to be dequeued at the coming clock edge. This scheme allows the entire pipeline to advance synchronously on each clock cycle. A stall in an intermediate pipeline stage causes all up-stream stages to stall at once. A caveat of this scheme is that this multi-stage feedback path could become the critical path, especially in a deeply pipelined design. In which case, one may want to break the feedback path at selected stages by using two-register-deep FIFOs with local flow control. A cyclic feedback path can also be broken by inserting two-register-deep FIFOs with local flow control.

### 6.5.2 GTECH RTL Analysis

Five TRSPEC descriptions are included in this analysis. The first three are the non-pipelined processor, two-stage pipelined processor and two-stage two-way superscalar processor, presented in Sections 6.2, 6.3 and 6.4, respectively. Two additional descriptions describe a three-stage pipelined processor and its corresponding two-way superscalar derivative. The three-stage pipelined processor corresponds to the datapath in Figure 6-3 with pipeline buffers inserted at both positions *S0* and *S1*. The three-stage pipelined processor description is derived from the two-stage pipelined

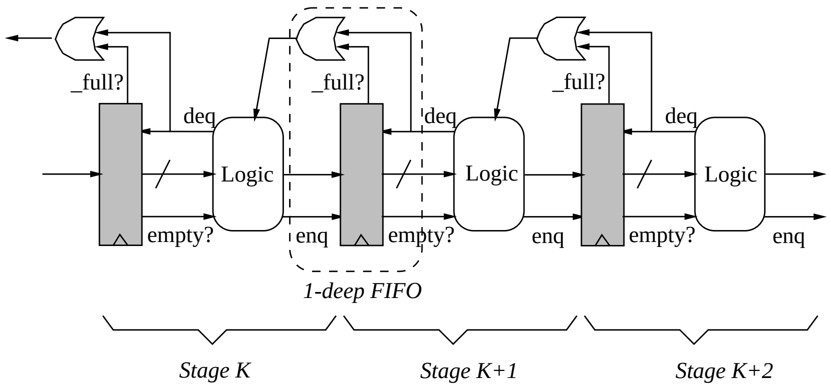


Figure 6-11: Synchronous pipeline with combinational multi-stage feedback flow control.

| | Unpipelined | 2-stage | 2-stage 2-way | 3-stage | 3-stage 2-way |
|---|---|---|---|---|---|
| | area ( % ) | area ( % ) | area ( % ) | area ( % ) | area ( % ) |
| Prog. Counter | 321 ( 7.4) | 321 ( 5.6) | 321 ( 4.0) | 321 ( 5.1) | 321 ( 3.4) |
| Reg. File | 1786 ( 41.2) | 1792 ( 31.1) | 2157 ( 26.9) | 1792 ( 28.1) | 2157 ( 22.7) |
| Mem. Interface | 981 ( 22.6) | 985 ( 17.1) | 985 ( 12.3) | 985 ( 15.4) | 985 ( 10.4) |
| ALU | 796 ( 18.3) | 796 ( 13.8) | 1588 ( 19.8) | 796 ( 12.5) | 1588 ( 16.7) |
| Pipe. Buffer(s) | 0 ( 0.0) | 737 ( 12.8) | 1858 ( 23.2) | 1305 ( 20.4) | 3342 ( 35.2) |
| Logic | 450 ( 10.4) | 1122 ( 19.5) | 1099 ( 13.7) | 1179 ( 18.4) | 1099 ( 11.6) |
| Total | 4334 (100.0) | 5753 (100.0) | 8008 (100.0) | 6378 (100.0) | 9492 (100.0) |
| Normalized Total | 1.00 | 1.33 | 18.5 | 1.47 | 2.19 |

*(Unit area = two-input* NAND *gate)*

Figure 6-12: Circuit area distributions for five processors.

processor descriptions following the same methodology presented in Section 6.3. All TRSPEC descriptions have been derived manually starting from the non-pipelined processor description and have been altered to reference external instruction and data memories through I/O ports.

The five TRSPEC processor descriptions are compiled to GTECH netlists for area and timing analyses by the *Synopsys RTL Analyzer*. The outputs of the *Synopsys RTL Analyzer* are tabulated in Figures 6-12 and 6-13. Figure 6-12 compares the amount of logic area needed by the five designs. (One unit of area corresponds to the area needed by a two-input *NAND* gate.) The total area increases 2.2 times going from a non-pipelined implementation to a three-stage two-way superscalar pipeline. As expected, both pipelining and superscalarity increase the pipeline buffer requirements (from 0 to 3342) and control logic requirements (from 450 to 1099). Superscalarity also doubles the ALU requirements and increases the register-file size because of the additional read and write ports.

The two tables in Figure 6-13 break down the delay of each processor's critical path into contributions by different parts of the processor. (The *Synopsys RTL Analyzer*

| | unpipelined | 2-stage | | 2-stage, 2-way | |
|---|---|---|---|---|---|
| | | Stage 1 | Stage 2 | Stage 1 | Stage 2 |
| Program Counter | *start* | *start* | | *start* | |
| Instruction Fetch | X | X | | X | |
| Operand Fetch | 4 | | | | |
| Raw Hazard | | 12 | | | |
| PC increment | | | | 18 | |
| *S1* | | 6 | *start* | 8 | *start* |
| 32-ALU | 20 | | 20 | | 20 |
| Write Back | 6 | | 5 | | 7 |
| Total | 30+X | 18+X | 25 | 26+X | 27 |
| if X=20 | 50 | 38 | 25 | 46 | 27 |

*(Unit delay = two-input NAND gate)*

| | 3-stage | | | 3-stage, 2-way | | |
|---|---|---|---|---|---|---|
| | Stage 1 | Stage 2 | Stage 3 | Stage 1 | Stage 2 | Stage 3 |
| Program Counter | *start* | | | *start* | | |
| Inst Fetch or PC Inc | X | | | X | | |
| *S0* | 6 | *start* | | 8 | *start* | |
| Instruction Decode | | 12 | | | 18 | |
| *S1* | | 8 | *start* | | 11 | *start* |
| 32-ALU | | | 20 | | | 23 |
| Write Back | | | 5 | | | 8 |
| Total | 6+X | 20 | 25 | 8+X | 29 | 31 |
| if X=20 | 26 | 20 | 25 | 28 | 29 | 31 |

*(Unit delay = two-input NAND gate)*

Figure 6-13: Critical path delays for five processors.

reports the logic delay in units that correspond to the propagation delay of a two-input *NAND* gate. The analysis does not consider load or fan-out.) For pipelined processors, separate critical paths are given for each stage. (A combinational path is assigned to a stage based on where the path starts). When a critical path involves an instruction fetch, X is used to represent the instruction memory lookup delay since the actual delay will vary with the instruction memory size and implementation. The critical path analysis does not consider the latencies of data memory operations since, if data memory latencies ever become a factor in the critical path, one should modify the synchronous data memory interface to spend more cycles rather than lengthening the cycle time.

The information generated by the *Synopsys RTL Analyzer* is helpful in deciding the effectiveness of various architectural features and modifications. Ideally, converting a non-pipelined microarchitecture to a $p$-stage pipelined microarchitecture should increase the clock frequency by $p$-fold, but this is rarely achieved in practice due to unbalanced partitioning and pipeline logic overhead. Assuming X is 20 time units, the two-stage pipelined processor only achieves a clock frequency that is 39% higher than the non-pipelined version. The three-stage pipeline processor achieves a 92% improvement.

Overall, the peak performance of the two-stage two-way superscalar processor is approximately twice that of the non-pipelined processor at approximately twice the cost in terms of area. The three-stage two-way superscalar processor appears to have the best performance-area trade-off since it has nearly 3 times the performance of the non-pipelined processors while consuming only 2.2 times more area.

A caveat in this analysis is that the results only give an indication of the processors' peak performance. The effects of instruction mix on the different microarchitectures must be analyzed by simulating the TRAC-generated RTL descriptions with benchmark applications. The combined feedback from circuit analysis and simulation can help steer the architect in an iterative exploration of a large number of architectural options and trade-offs.

## 6.6 Summary

This chapter describes how to generate a pipelined processor design from an ISA specification by source-to-source transformations in the TRSpec operation-centric framework. Transformations to create superscalar designs are also possible. Currently, the transformations described in this chapter have to be performed manually. An effort to semi-automate the process is underway [Lis00, Ros00]. In the future, the mechanics of the transformation would be automated, but human intervention would still be required to guide these transformations at a high level, such as selecting the degree of superscalarity and the positions of the pipeline stages.

# Chapter 7

# Conclusions

This thesis creates a new operation-centric hardware development framework that employs an easier and more intuitive hardware design abstraction.

## 7.1 Summary of Work

**Operation-Centric Hardware Description:** This thesis presents the concepts and advantages of operation-centric hardware description where the behavior of a system is decomposed and described as a collection of operations. Typically, an operation is defined by a predicate condition and an effect. An operation affects the state of the system globally and atomically. This atomic semantics simplifies the task of hardware description by permitting the designer to formulate each operation as if the system were otherwise static.

**TRSpec Hardware Description Language:** This thesis presents TRSpec, an adaptation of Term Rewriting Systems (TRS) for operation-centric hardware description. This synthesizable TRS language includes extensions beyond the standard TRS formalism to increase its compactness and expressiveness in hardware description. On the other hand, in some areas, the generality of TRS has to be restricted with the help of a type system to ensure a description's synthesizability into a finite-state machine. Specifically, TRSpec disallows infinite-size terms and also disallows rules that can change the size of the terms.

**Abstract Transition Systems:** This thesis defines an Abstract Transition System (ATS). ATS is an operation-centric state machine abstraction. ATS is a convenient intermediate representation when mapping TRSpec, or other source-level operation-centric languages, to hardware implementations. ATS has all of the essential properties of an operation-centric framework but without the syntactic complications of a source-level language. Generalized synthesis and optimization algorithms can be developed in the ATS abstraction, independent of source language variations.

**Hardware Synthesis and Scheduling:** This thesis develops the theories and algorithms necessary to create an efficient hardware implementation from an operation-centric description. In particular, this thesis explains how to implement an ATS as a synchronous finite state machine. The crux of the synthesis problem involves finding a valid composition of the ATS transitions in a coherent state transition system that carries out as many ATS transitions concurrently as possible. This thesis presents both a straightforward reference implementation and two optimized implementations.

**Term Rewriting Architectural Compiler:** The ideas in this thesis are realized in the Term Rewriting Architectural Compiler (TRAC), a TRSPEC-to-Verilog compiler. Design exercises employing TRSPEC and TRAC have shown that an operation-centric hardware development framework offers a significant reduction in design time and effort when compared to traditional frameworks, without loss in implementation quality.

**Microarchitectural Transformations:** This thesis investigates the application of TRSPEC and TRAC to the design of pipelined superscalar processors. The design flow starts from a basic instruction set architecture given in TRSPEC and progressively incorporates descriptions of pipelining and superscalar mechanisms as source-to-source transformations.

## 7.2 Future Work

This thesis is a preliminary investigation into operation-centric frameworks for hardware development. The results of this thesis provide a starting point for this research. This section points out the weaknesses in this thesis and proposes possible resolutions in follow-on research. This section also describes work stemming from applying the technologies in this thesis.

### 7.2.1 Language Issues

The TRSPEC language supports the essential aspects of operation-centric hardware description. However, in many respects, TRSPEC is a rudimentary language without many of the common features in modern programming languages. Syntactically, TRSPEC borrows from the TRS notation, which is not suited for describing large or modular designs. These and many other issues are being addressed by the development of BlueSpec, a new operation-centric hardware description language [Aug00]. Like TRSPEC, BlueSpec is a strongly typed language supporting algebraic types. BlueSpec is semantically similar to TRSPEC, but it borrows from the syntax and features of the Haskell [JHA+98] functional language. The salient features of BlueSpec are outlined below.

**Compact Syntax:** A TRSPEC rule equates to a BlueSpec function of state-to-state. A BlueSpec function also supports the use of pattern matching to specify the firing condition of a rule. To reduce repetition, each pattern can have multiple "right-hand-side" effect terms guarded by different predicate expressions. Semantically, these are different rules that happen to have the same left-hand-side pattern. To reduce verbosity, BlueSpec also uses a "named" record notation for product and disjunct terms. Thus, a product pattern can constrain only specific fields by name, without mentioning the unconstrained fields. A right-hand-side effect term can also specify changes for only selected fields without mentioning the unaffected fields.

**Complex Expressions:** TRSpec, as presented in this thesis, only supports simple arithmetic and logical expressions. The TRAC implementation of TRSpec actually allows more complicated expressions like *if-then-else* statements and *switch-case* statements on both scalar and algebraic types. BlueSpec further allows the full range of expression constructs from the Haskell language. In particular, BlueSpec supports the use of functions that can be compiled into pure combinational logic.

**Generalized Abstract Types:** Besides built-in arrays and FIFOs, BlueSpec allows user-defined abstract types with an arbitrary list of combinational and state-transforming interfaces. External library modules can be incorporated into a TRSpec description as a custom abstract type.

**Additional I/O Types:** The simple I/O mechanisms of TRSpec are not enough to meet all design scenarios efficiently. BlueSpec supports additional I/O type classes to give users additional options over the exact implementation of I/O interfaces. The additional I/O type classes are useful when designing an I/O interface to meet a pre-defined synchronous handshake. BlueSpec also provides I/O constructs for combining modular BlueSpec designs.

### 7.2.2 Synthesis Issues

The synthesis algorithm in this thesis always maps the entire effect of an operation (i.e., a TRSpec rule) into a single clock cycle. The scheduling algorithm always attempts to maximize hardware concurrency. These two simplifying policies are necessarily accompanied by the assumption that any required hardware resources are available. In practice, TRAC's implementation of these policies, in conjunction with good register-transfer-level (RTL) logic optimizations, results in reasonably efficient implementations. Nevertheless, the synthesis and scheduling algorithms can be improved in many dimensions. Some of the optimizations outlined below are already part of the normal RTL compilation that takes place after TRAC. However, there is an advantage to incorporate these optimizations into the front-end compilation because front-end compilers like TRAC have better access to high-level semantics embedded in the source descriptions. For example, TRAC performs its own RTL optimizations before generating its output. TRAC can trace two RTL signals to their usage in the source description, and if the two signals originate from two conflicting rules then TRAC can conclude the signals are never used simultaneously. The same inference would be hard or impossible when the same design is passed to the back-end compiler in a distilled RTL format.

**Technology Library Mapping:** TRAC can instantiate library modules that have been declared explicitly as user-defined abstract types, but TRAC cannot map an arbitrary TRSpec description to target a specific set of library macros. The RTL descriptions generated by TRAC only assume three state primitives: registers, arrays and FIFOs. The only combinational logic primitives used by TRAC are multiplexers. The remaining combinational logics are expressed as Verilog expressions. Normally, TRAC can defer technology-specific mappings to back-end RTL compilers like Synopsys. However, unlike gate-array or standard-cell implementations, the quality of FPGA synthesis is very sensitive to the use of vender-specific RTL coding guidelines and library macros. It is important for TRAC to generate optimized FPGA implementations because a potential application of TRAC and TRSpec is to facilitate rapid creation of hardware prototypes using reconfigurable technologies.

**Mapping Operations to Multiple Clock Cycles:** In many hardware applications, there are hard limits on the amount and the type of hardware resources available. In other cases, factors like power consumption and manufacturability place greater emphasis on lower hardware utilization over absolute performance. Under these assumptions, it is not optimal, sometimes even unrealistic, to require the effect of an operation to always execute in one clock cycle. For example, an operation may perform multiple reads and writes to the same memory array whereas the implementation technology does not permit multi-ported memory. Also, it may not make sense to instantiate multiple floating-point multipliers only because one of the operations performs multiple floating-point multiplications. Finally, some operations, like divide, simply cannot be reasonably carried out in a single clock cycle. Currently, there is an effort to develop a new method where the effect of an operation can be executed over multiple clock cycles to meet resource requirements [Ros00]. The current approach partitions a complex operation into multiple smaller operations that are suitable for single-cycle mapping. The key aspect in this transformation is to add appropriate interlocks such that the atomicity of the original operations are mimicked by the execution of the partitioned operations over multiple clock cycles. The synthesis algorithms in this thesis are directly applicable to the transformed system.

**Automatic Pipelining and Superscalar Transformations:** Chapter 6 of this thesis describes manual source-to-source transformations for creating superscalar and pipelined processors. Follow-on efforts are looking into automating these transformations. The steps to automate the pipelining transformation are related to the partitioning discussed in the previous paragraph. For pipelining, a single TRSpec rule is partitioned for execution over multiple clock cycles. In the context of pipelining, besides maintaining the atomic semantics of the original rules, there is added attention to create sub-rules that can be executed in a pipelined fashion [Ros00]. To automate superscalar transformations, sub-rules in the same pipeline stage are identified and syntactically composed to form new superscalar rules [Lis00]. In a related effort, rule transformations are applied to the verification of pipelined superscalar processors. For example, using rule composition, it is possible to reduce a pipelined processor to a more easily verified non-pipelined equivalent by eliminating pipeline buffers one stage at a time [Lis00].

**Power, Area and Timing-Aware Synthesis:** The current implementation of TRAC chiefly focuses on generating a correct RTL implementation for operation-centrically specified behaviors. The RTL implementations are optimized with a single-minded goal to maximize hardware concurrency. In many applications, it is necessary to optimize for other factors like power, area and timing. Currently, power, area and timing analyses are available during the RTL compilation phase. The designer can modify the source description according to the feedback from RTL synthesis. The three improvements to TRAC discussed above (technology library mapping, multi-cycle operations, and pipelining transformation) open up the possibility to automate this design refinement process. Incorporating technology-specific library mapping into the front-end enables TRAC to estimate power, area and timing early on in the synthesis. Thus, TRAC can adjust its optimization goals accordingly. To meet a specific power or area budget, TRAC can partition an operation over multiple clock cycles to reuse hardware resources. To meet a specific timing requirement, pipelining transformation can be employed to break up the critical path.

### 7.2.3 Applications

This thesis presents several processor-related examples. Although TRSpec and TRAC are good architectural development tools, their applications have a much larger domain. The following paragraphs point out some of the applications currently being explored.

**Reconfigurable Computing:** Given the current pace of development in reconfigurable computing, it is likely that some day all personal computers will be shipped with a user-reconfigurable hardware alongside of the CPU. The high-level abstraction of the TRSpec framework lowers the effort and expertise required to develop hardware *applets* for the reconfigurable coprocessing hardware. A programmer could retarget part of a software application for hardware implementation using the same level of time and effort as software development. Even today, when combined with suitable reconfigurable technologies like the Annapolis Wildcard, TRSpec and TRAC already can provide an environment where the reconfigurable hardware can be used as software accelerators (see discussions in Section 5.4).

**Hardware Cache Coherence:** Operation-centric descriptions based on TRS have been applied to the study of memory consistency and cache coherence. In CACHET, TRS is used to formally specify a dynamically adaptive cache coherence protocol for distributed shared-memory systems [SAR99a]. The Commit-Reconcile and Fences

(CRF) memory model uses TRS to capture the semantics of elemental memory operations in a novel memory model designed for modern out-of-order superscalar microarchitectures [SAR99b]. The properties of these formally specified memory consistency and coherence models can be verified using theorem proving techniques as well as by simulating against a reference TRS specification. Recent efforts have attempted to couple synthesis and formal verification to the same source description. The goal is to capture architectural-level TRS models in TRSpec for both formal verification and automatic synthesis into memory controllers and cache-coherence engines.

**Microarchitecture Research:** With TRSpec and TRAC, a high-density field programmable hardware platform becomes a powerful hardware prototyping testbench. Such a prototyping framework can enable VLSI-scale "bread boarding" such that even a small research group can explore a variety of architectural ideas quickly and spontaneously. Studying prototypes can expose subtle design and implementation issues that are too easily overlooked on paper or in a simulator. A prototype of a highly concurrent system also delivers much higher execution rates than simulation. A thorough investigation using a hardware prototype lends much greater credence to experimental research of revolutionary ideas.

**Teaching:** A high-level operation-centric framework is also a powerful teaching aide. In a lecture, an operation-centric TRSpec description gives an intuitive functional explanation. An operation-centric description also allows digital-design issues to be separated from architectural ones. The high-level architectural prototyping environment discussed in the previous paragraph can also be integrated into a computer architecture course where an advanced hardware student can study a broad range of architectural issues in a hands-on manner. In addition to the materials presented in class, a student can acquire an even deeper understanding of a mechanism by tinkering with its high-level description and study the effects on a simulator or a prototyping platform. This kind of independent exercise will help students build stronger intuitions for solving architectural problems. On the other hand, the course's emphasis on mechanisms rather than implementation also makes it ideal for software students who simply want to understand how to use the complex features of modern systems.

## 7.3 Concluding Remarks

In the short term, a high-level operation-centric hardware development framework cannot completely replace current RTL-based design practices. Clearly, there is a class of applications, such as microprocessors, that demands the highest possible performance and has the economic incentives to justify the highest level of development effort and time. Nevertheless, a steady industry-wide move toward a higher-level design environment is inevitable. When the integrated-circuit design complexity surpassed one million gates in the early 90's, designers abandoned schematic capture in favor of textual hardware description languages. An analogous evolution to a still higher-level design environment is bound to repeat when the complexity of integrated-circuit designs exceeds the capacity of current design tools.

Ultimately, the goal of a high-level description is to provide an uncluttered design representation that is easy for a designer to comprehend and reason about. Although a concise notation is helpful, the utility of a "high-level" description framework has to come from the elimination of some "lower-level" details. It is in this sense that an operation-centric framework can offer an advantage over traditional RTL design frameworks. Any non-trivial hardware design consists of multiple concurrent threads of computation in the form of concurrent finite state machines. This concurrency must be managed explicitly in traditional representations. In an operation-centric description, parallelism and concurrency are implicit in the source-level descriptions, only to be discovered and managed by an optimizing compiler.

# Appendix A

# TRSpec Language Syntax

## A.1 Keywords

### A.1.1 Keywords in Type Definitions

‘`Type`’: start of an algebraic type definition
‘`IType`’: start of an input port type definition
‘`OType`’: start of an output port type definition
‘`TypeSyn`’: start of a type synonym definition

‘`Bit`’: declaring a built-in unsigned integer type
‘`Int`’: declaring a built-in signed integer type
‘`Bool`’: declaring a built-in Boolean type
‘`Array`’: declaring a built-in abstract array type
‘`Fifo`’: declaring a built-in abstract FIFO type

### A.1.2 Keywords in Rule and Source Term Declarations

‘`Rule`’: start of a rule declaration
‘`Init`’: start of a source term declaration
‘`if`’: start of a predicate expression
‘`where`’: start of a LHS or RHS where binding list

## A.2 TRS

***TRS*** :: ***TypeDefinitions Rules SourceTerm***

## A.3 Type Definitions

***TypeDefinitions*** :: ***TypeDefinition***
▯ ***TypeDefinition TypeDefinitions***

***TypeDefinition*** :: ***DefineBuiltInType***
▯ ***DefineAlgebraicType***
▯ ***DefineAbstractType***
▯ ***DefineIoType***
▯ ***DefineTypeSynonym***

### A.3.1 Built-In Type

***DefineBuiltInType*** :: `Type` ***TypeName*** `=` `Bit[`***BitWidth***`]`
▯ `Type` ***TypeName*** `=` `Int[`***BitWidth***`]`
▯ `Type` ***TypeName*** `=` `Bool`

### A.3.2 Algebraic Type

***DefineAlgebraicType*** :: `Type` ***TypeName*** `=` ***AlgebraicType***

***AlgebraicType*** :: ***ProductType***
▯ ***SumType***

#### Product Types

***ProductType*** :: ***ConstructorName***(***TypeName***$_1$, ..., ***TypeName***$_k$)
*Note: where* $k \geq 1$

#### Sum Types

***SumType*** :: ***Disjuncts***
***Disjuncts*** :: ***Disjunct***
▯ ***Disjunct*** `||` ***Disjuncts***
***Disjunct*** :: ***ConstructorName***(***TypeName***$_1$, ..., ***TypeName***$_k$)
*Note: where* $k \geq 0$

### A.3.3 Abstract Type

***DefineAbstractType*** :: `Type` ***TypeName*** `=` ***AbstractType***

***AbstractType*** :: `Array [`***TypeName***$_{index}$`]` ***TypeName***$_{data}$
▯ `Fifo` ***TypeName***

### A.3.4 I/O Type

| | | |
|---|---|---|
| ***DefineIoType*** | :: | `IType` ***TypeName*** `=` ***TypeName*** |
| | [] | `OType` ***TypeName*** `=` ***TypeName*** |

### A.3.5 Type Synonym

| | | |
|---|---|---|
| ***DefineTypeSynonym*** | :: | `TypeSyn` ***TypeName*** `=` ***TypeName*** |

### A.3.6 Miscellaneous

| | | |
|---|---|---|
| ***BitWidth*** | :: | [1-9][0-9]* |
| ***TypeName*** | :: | [A-Z][A-Z0-9]* |
| ***ConstructorName*** | :: | [A-Z][a-z0-9]+ |

## A.4 Rules

| | | |
|---|---|---|
| ***Rules*** | :: | ***Rule*** |
| | [] | ***Rule Rules*** |
| ***Rule*** | :: | `Rule` ***RuleName LHS*** `==>` ***RHS*** |
| | | *Note: The main pattern in **LHS** and the main expression in **RHS** must have the same type* |

### A.4.1 Left Hand Side

| | | |
|---|---|---|
| ***LHS*** | :: | ***Pattern*** |
| | [] | ***Pattern PredicateClause*** |
| | [] | ***Pattern LhsWhereClause*** |
| | [] | ***Pattern PredicateClause LhsWhereClause*** |
| ***Pattern*** | :: | '`-`' [] ***VariableName*** [] ***NumericalConstant*** |
| | [] | ***ConstructorName***(***Pattern***$_1$, ..., ***Pattern***$_k$) |
| | | *Note: where* $k \geq 0$ |
| ***PredicateClause*** | :: | `if` ***Expression*** |
| | | *Note: **Expression** must have an integer type* |
| ***LhsWhereClause*** | :: | `where` ***PatternMatches*** |
| ***PatternMatches*** | :: | ***PatternMatch*** |
| | [] | ***PatternMatch PatternMatches*** |
| ***PatternMatch*** | :: | ***Pattern*** `=` ***Expression*** |
| | | *Note: **Pattern** and **Expression** must have the same type* |

### A.4.2 Right Hand Side

| | | |
|---|---|---|
| ***RHS*** | :: | ***Expression*** |
| | [] | ***Expression RhsWhereClause*** |

| | | |
|---|---|---|
| ***RhsWhereClause*** | :: | `where` ***Bindings*** |
| ***Bindings*** | :: | ***Binding*** |
| | [] | ***Binding Bindings*** |
| ***Binding*** | :: | ***VariableName*** `=` ***Expression*** |

### A.4.3 Expressions

| | | |
|---|---|---|
| ***Expression*** | :: | '`-`' [] ***VariableName*** [] ***NumericalConstant*** |
| | [] | ***ConstructorName***(***Expression***$_1$, ..., ***Expression***$_k$) |
| | [] | ***PrimitiveOp***(***Expression***$_1$, ..., ***Expression***$_k$) |
| | | *Note: Infix representation of arithmetic and logical operations is supported as syntactic sugar* |
| | [] | ***AbsInterface*** |

| | | |
|---|---|---|
| ***PrimitiveOp*** | :: | ***Arithmetic*** [] ***Logical***[] ***Relational*** |
| ***Arithmetic*** | :: | `Add` [] `Sub` [] `Multiply` [] `Divide` [] `Mod` [] `Negate` |
| ***Logical*** | :: | `Not` [] `And` [] `Or` |
| | [] | `BitwiseNegate` [] `BitwiseAnd` [] `BitwiseOr` [] `BitwiseXor` |
| ***Relational*** | :: | `Equal` [] `NotEqual` [] `GreaterThan` [] `GreaterThanEqualTo` |
| | [] | `LessThan` [] `LessThanEqualTo` |

| | | |
|---|---|---|
| ***AbsInterface*** | :: | ***Expression***$_{array}$`.read(`***Expression***$_{idx}$`)` |
| | | *also as:* ***Expression***$_{array}$`[`***Expression***$_{idx}$`]` |
| | [] | ***Expression***$_{array}$`.write(`***Expression***$_{idx}$ `,` ***Expression***$_{data}$`)` |
| | | *also as:* ***Expression***$_{array}$`[`***Expression***$_{idx}$`:=`***Expression***$_{data}$`]` |
| | [] | ***Expression***$_{fifo}$`.first()` |
| | [] | ***Expression***$_{fifo}$`.enq(`***Expression***`)` |
| | [] | ***Expression***$_{fifo}$`.deq()` |
| | [] | ***Expression***$_{fifo}$`.clear()` |

### A.4.4 Miscellaneous

| | | |
|---|---|---|
| ***RuleName*** | :: | "[A-Za-z0-9 ,.:?]+" |
| ***VariableName*** | :: | [a-z][a-z0-9]* |

## A.5 Source Term

| | | |
|---|---|---|
| ***SourceTerm*** | :: | `Init` ***Expression*** |
| | [] | `Init` ***Expression RhsWhereClause*** |

# Appendix B

# TRSpec Description of a MIPS Processor

## B.1 Type Definitions

### B.1.1 Processor States

```
Type        PROC = Proc(PC_O,RF,BD,BE,BM,BW,IPORT,DPORT_R,DPORT_W,SHIFTER)
```

#### User Visible Registers

```
OType       PC_O = PC
Type          PC = Bit[32]
Type          RF = Array [RNAME] VAL
Type       RNAME = Reg0 || Reg1 || Reg2 || Reg3
                || Reg4 || Reg5 || Reg6 || Reg7
                || Reg8 || Reg9 || Reg10 || Reg11
                || Reg12 || Reg13 || Reg14 || Reg15
                || Reg16 || Reg17 || Reg18 || Reg19
                || Reg20 || Reg21 || Reg22 || Reg23
                || Reg24 || Reg25 || Reg26 || Reg27
                || Reg28 || Reg29 || Reg30 || Reg31
Type         VAL = Bit[32]
```

#### Pipeline Stage Buffers

```
Type          BD = Fifo BD_TEMPLATE
Type BD_TEMPLATE = BdTemp(PC,INST)
TypeSyn       BE = BS
TypeSyn       BM = BS
TypeSyn       BW = BS
```

```
ABSType        BS = enq(BS_TEMPLATE)
                 || deq()
                 || clear()
                 || isdest(RNAME) BOOL
                 || forward(RNAME) VAL
                 || canforward(RNAME) BOOL
                 || first() BS_TEMPLATE
                 || notempty() BOOL
                 || notfull() BOOL
Type         BOOL = False || True
```

*Note: For readability a disjunct term without any subterms, such as* `True()`*, can be written as the constructor name alone without being followed by parentheses.*

```
Type  BS_TEMPLATE = BsTemp(PC,I_TEMPLATE)
Type   I_TEMPLATE = Itemp(MINOROP,WBACK,READY,RD,VAL,VAL,VAL)
Type      MINOROP = MAdd || MAddu || MAnd || MSub
                 || MSubu || MNor || MOr || MXor
                 || MSlt || MSll || MSra || MSrl
                 || MLoad || MStore || MWback || MNop
                 || MOnemore
TypeSyn     WBACK = BOOL
TypeSyn     READY = BOOL
```

### Barrel Shifter

```
Type       AMOUNT = Bit[5]
Type         LEFT = Right || Left
Type        ARITH = Logical || Arith
ABSType   SHIFTER = shift(AMOUNT,LEFT,ARITH,VAL) VAL
```

### Input and Output

```
IType       IPORT = INST
Type      DPORT_R = DportR(RADDR,RDATA)
Type      DPORT_W = DportW(WADDR,WDATA,WVALID)
OType       IADDR = PC
IType       IDATA = INST
SOType      RADDR = ADDR
IType       RDATA = VAL
Type         ADDR = Bit[32]
SOType      WADDR = ADDR
SOType      WDATA = VAL
SOType     WVALID = BOOL
```

## B.1.2 Instruction Set Architecture

```
Type         INST = Mips(OP,RD,RS,RT,SA,FUNC)
Type           OP = Special || Bcond || Jj || Jal
                  || Beq || Bne || Blez || Bgtz
                  || Addi || Addiu || Slti || Sltiu
                  || Andi || Ori || Xori || Lui
                  || Cop0 || Cop1 || Cop2 || Cop3
                  || Op24 || Op25 || Op26 || Op27
                  || Op30 || Op31 || Op32 || Op33
                  || Op34 || Op35 || Op36 || Op37
                  || Lb || Lh || Lwl || Lw
                  || Lbu || Lhu || Lwr || Op47
                  || Sb || Sh || Swl || Sw
                  || Op54 || Op55 || Swr || Op57
                  || Lwc0 || Lwc1 || Lwc2 || Lwc3
                  || Op64 || Op65 || Op66 || Op67
                  || Swc0 || Swc1 || Swc2 || Swc3
                  || Op74 || Op75 || Op76 || Op77

TypeSyn        RD = RNAME
TypeSyn        RS = RNAME
TypeSyn        RT = RNAME

Type           SA = Bit[5]

Type         FUNC = Sll || Func01 || Srl || Sra
                  || Sllv || Func05 || Srlv || Srav
                  || Jr || Jalr || Func12 || Func13
                  || SysCall || Break || Func16 || Func17
                  || Mfhi || Mthi || Mflo || Mtlo
                  || Func24 || Func25 || Func26 || Func27
                  || Mult || Multu || Div || Divu
                  || Func34 || Func35 || Func36 || Func37
                  || Add || Addu || Sub || Subu
                  || And || Or || Xor || Nor
                  || Func50 || Func51 || Slt || Sltu
                  || Func54 || Func55 || Func56 || Func57
                  || Func60 || Func61 || Func62 || Func63
                  || Func64 || Func65 || Func66 || Func67
                  || Func70 || Func71 || Func72 || Func73
                  || Func74 || Func75 || Func76 || Func77

Type        BCOND = Bltz || Bgez || Bcond02 || Bcond03
                  || Bcond04 || Bcond05 || Bcond06 || Bcond07
                  || Bcond10 || Bcond11 || Bcond12 || Rim13
                  || Bcond14 || Bcond15 || Bcond16 || Bcond17
                  || Bltzal || Bgezal || Bcond22 || Bcond23
                  || Bcond24 || Bcond25 || Bcond26 || Bcond27
                  || Bcond30 || Bcond31 || Bcond32 || Rim33
                  || Bcond34 || Bcond35 || Bcond36 || Bcond37
```

## B.2 Rules

### B.2.1 M4 Macros

```
define(‘STALL’,‘((be.isdest($1)&&!be.canforward($1)) ||
                 (bm.isdest($1)&&!bm.canforward($1)) ||
                 (bw.isdest($1)&&!bw.canforward($1)))’)

define(‘FORWARD’,‘(be.canforward($1)?
                   be.forward($1):
                   (bm.canforward($1)?
                    bm.forward($1):
                    (bw.canforward($1)?bw.forward($1):rf[$1])))’)
```

### B.2.2 Fetch Stage Rules

```
Rule "Instruction Fetch and Speculate"
     Proc(pc,rf,bd,be,bm,bw,inst,rport,wport,shftr)
          if  bd.notfull()
==>  Proc(pc+4,rf,bd.enq(BdTemp(pc,inst)),be,bm,bw,
               inst,rport,wport,shftr)
```

## B.2.3 Decode Stage Rules

### I-Type Instructions

```
Rule "Decode Immediate"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
         if  canhandle && !STALL(rs)
         &&  bd.notempty() && be.notfull()
      where  BdTemp(pc',inst) = bd.first()
             Mips(op,rs,rt,immh,immm,imml) = inst
             imm16 = {immh,immm,imml}
             canhandle = (op==Addi) || (op==Addiu) ||
                         (op==Slti) || (op==Sltiu) ||
                         (op==Andi) || (op==Ori) ||
                         (op==Xori)
==>  Proc(pc,rf,bd.deq(),be',bm,bw,iport,rport,wport,shftr)
      where  be' = be.enq(BsTemp(pc',itemp))
             itemp = (rt==Reg0)?
                        Itemp(MNop,False,False,-,-,-,-):
                        Itemp(mop,True,False,rt,vs,vimm,-)
             vs = FORWARD(rs)
             vimm = {(immh[4:4]?16'hffff:16'h0000),imm16[15:0]}
             mop = switch(op)
                        case Addi:  MAdd
                        case Addiu:  MAdd
                        case Slti:  MSlt
                        case Sltiu:  MSlt
                        case Andi:  MAnd
                        case Ori:  MOr
                        case Xori:  MXor
Rule "Instruction Decode Lui"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
         if  bd.notempty() && be.notfull()
      where  BdTemp(pc',inst) = bd.first()
             Mips(Lui,rd,rs,immh,immm,imml) = inst
             imm16 = {immh,immm,imml}
==>  Proc(pc,rf,bd.deq(),be',bm,bw,iport,rport,wport,shftr)
      where  be' = be.enq(BsTemp(pc',itemp))
             itemp = (rd==Reg0)?
                        Itemp(MNop,False,False,-,-,-,-):
                        Itemp(MWback,True,True,rd,{imm16[15:0],16'b0},-,-)
```

```
Rule "Instruction Decode Load"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
          if  canhandle && !STALL(base)
          &&  bd.notempty() && be.notfull()
      where  BdTemp(pc',inst) = bd.first()
             Mips(op,base,rt,offh,offm,offl) = inst
             offset = {(offh[4:4]?(16'hffff):16'h0), offh,offm,offl}
             canhandle = (op==Lw)
==>  Proc(pc,rf,bd.deq(),be',bm,bw,iport,rport,wport,shftr)
      where  be' = be.enq(BsTemp(pc',itemp))
             itemp = (rt==Reg0)?
                       Itemp(MNop,False,False,-,-,-,-):
                       Itemp(mop,True,False,rt,vbase,offset,-)
             vbase = FORWARD(base)
             mop = switch(op)
                       case Lw:  MLoad

Rule "Instruction Decode Store"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
          if  canhandle && !(STALL(base) || STALL(rt))
          &&  bd.notempty() && be.notfull()
      where  BdTemp(pc',inst) = bd.first()
             Mips(op,base,rt,offh,offm,offl) = inst
             offset = {(offh[4:4]?(16'hffff):16'h0),offh,offm,offl}
             canhandle = (op==Sw)
==>  Proc(pc,rf,bd.deq(),be',bm,bw,iport,rport,wport,shftr)
      where  be' = be.enq(BsTemp(pc',
                            Itemp(mop,False,False,-,vbase,offset,vt)))
             vbase = FORWARD(base)
             vt = FORWARD(rt)
             mop = switch(op)
                       case Sw:  MStore

Rule "Decode R-Compare Branch Taken"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
          if  canhandle && !(STALL(rs) || STALL(rt)) && taken
          &&  bd.notempty()
      where  BdTemp(pc',inst) = bd.first()
             Mips(op,rs,rt,offh,offm,offl) = inst
             offset = {offh,offm,offl}
             canhandle = (op==Beq) || (op==Bne)
==>  Proc(target,rf,bd.clear(),be,bm,bw,iport,rport,wport,shftr)
      where  vs = FORWARD(rs)
             vt = FORWARD(rt)
             voff = {(offh[4:4]?(14'h3fff):(14'h0000)),(offset[15:0]),2'b00}
             taken = switch(op)
                       case Beq:(vs==vt)
                       case Bne:(vs!=vt)
             target = pc'+32'h4+voff
```

```
Rule "Decode R-Compare Branch Not-Taken"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
         if  canhandle && !(STALL(rs) || STALL(rt)) && !taken
         &&  bd.notempty()
      where  BdTemp(pc',inst) = bd.first()
             Mips(op,rs,rt,offh,offm,offl) = inst
             offset = {offh,offm,offl}
             canhandle = (op==Beq) || (op==Bne)
==>  Proc(pc,rf,bd.deq(),be,bm,bw,iport,rport,wport,shftr)
      where  vs = FORWARD(rs)
             vt = FORWARD(rt)
             taken = switch(op)
                       case Beq:(vs==vt)
                       case Bne:(vs!=vt)
Rule "Decode Compare-To-Zero Branch Taken"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
         if  canhandle && !STALL(rs) && taken
         &&  bd.notempty()
      where  BdTemp(pc',inst) = bd.first()
             Mips(op,rs,-,offh,offm,offl) = inst
             offset = {offh,offm,offl}
             canhandle = (op==Blez) || (op==Bgtz)
==>  Proc(target,rf,bd.clear(),be,bm,bw,iport,rport,wport,shftr)
      where  vs = FORWARD(rs)
             voff = {(offh[4:4]?(14'h3fff):(14'h0000)),(offset[15:0]),2'b00}
             taken = switch(op)
                       case Blez:(vs[31:31]) || (vs==0)
                       case Bgtz:(!vs[31:31]) || (vs!=0)
             target = pc'+32'h4+voff
Rule "Decode Compare-To-Zero Branch Not-Taken"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
         if  canhandle && !STALL(rs) && !taken
         &&  bd.notempty()
      where  BdTemp(pc',inst) = bd.first()
             Mips(op,rs,-,offh,offm,offl) = inst
             offset = {offh,offm,offl}
             canhandle = (op==Blez) || (op==Bgtz)
==>  Proc(pc,rf,bd.deq(),be,bm,bw,iport,rport,wport,shftr)
      where  vs = FORWARD(rs)
             taken = switch(op)
                       case Blez:(vs[31:31]) || (vs==0)
                       case Bgtz:(!vs[31:31]) || (vs!=0)
```

```
Rule "Decode Bcond Branch Taken"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
         if  !(STALL(rs)) && taken
         &&  bd.notempty()
      where  BdTemp(pc',inst) = bd.first()
             Mips(Bcond,rs,type,offh,offm,offl) = inst
             offset = {offh,offm,offl}
             canhandle = (type==Bltz) || (type==Bgez)
==>  Proc(target,rf,bd.clear(),be,bm,bw,iport,rport,wport,shftr)
      where  vs = FORWARD(rs)
             voff = {(offh[4:4]?(14'h3fff):(14'h0000)),(offset[15:0]),2'b00}
             taken = switch(type)
                        case Bltz:(vs[31:31])
                        case Bgez:(!vs[31:31])
             target = pc'+32'h4+voff
Rule "Decode Bcond Branch Not-Taken"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
         if  !(STALL(rs)) && !taken
         &&  bd.notempty()
      where  BdTemp(pc',inst) = bd.first()
             Mips(Bcond,rs,type,offh,offm,offl) = inst
             offset = {offh,offm,offl}
             canhandle = (type==Bltz) || (type==Bgez)
==>  Proc(pc,rf,bd.deq(),be,bm,bw,iport,rport,wport,shftr)
      where  vs = FORWARD(rs)
             voff = {(offh[4:4]?(14'h3fff):(14'h0000)),(offset[15:0]),2'b00}
             taken = switch(type)
                        case Bltz:(vs[31:31])
                        case Bgez:(!vs[31:31])
Rule "Decode Bcond Branch-and-Link Taken"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
         if  !(STALL(rs)) && taken
         &&  bd.notempty() && be.notfull()
      where  BdTemp(pc',inst) = bd.first()
             Mips(Bcond,rs,type,offh,offm,offl) = inst
             offset = {offh,offm,offl}
             canhandle = (type==Bltzal) || (type==Bgezal)
==>  Proc(target,rf,bd.clear(),be',bm,bw,iport,rport,wport,shftr)
      where  vs = FORWARD(rs)
             voff = {(offh[4:4]?(14'h3fff):(14'h0000)),(offset[15:0]),2'b00}
             taken = switch(type)
                        case Bltzal:(vs[31:31])
                        case Bgezal:(!vs[31:31])
             be' = be.enq(BsTemp(pc',
                              Itemp(MWback,True,True,Reg31,pc'+8,-,-)))
             target = pc'+32'h4+voff
```

```
Rule "Decode Bcond Branch-and-Link Not-Taken"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
         if  !(STALL(rs)) && !taken
         &&  bd.notempty()
      where  BdTemp(pc',inst) = bd.first()
             Mips(Bcond,rs,type,offh,offm,offl) = inst
             offset = {offh,offm,offl}
             canhandle = (type==Bltzal) || (type==Bgezal)
==>  Proc(pc,rf,bd.deq(),be,bm,bw,iport,rport,wport,shftr)
      where  vs = FORWARD(rs)
             voff = {(offh[4:4]?(14'h3fff):(14'h0000)),(offset[15:0]),2'b00}
             taken = switch(type)
                        case Bltzal:(vs[31:31])
                        case Bgezal:(!vs[31:31])
```

### J-Type Instructions

```
Rule "Instruction Decode J"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
         if  bd.notempty()
      where  BdTemp(pc',inst) = bd.first()
             Mips(Jj,off5,off4,off3,off2,off1) = inst
             offset = {off5,off4,off3,off2,off1}
==>  Proc(target,rf,bd.clear(),be,bm,bw,iport,rport,wport,shftr)
      where  target = {pc'[31:28],offset[25:0],2'b00}
Rule "Instruction Decode Jal"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
         if  bd.notempty() && be.notfull()
      where  BdTemp(pc',inst) = bd.first()
             Mips(Jal,off5,off4,off3,off2,off1) = inst
             offset = {off5,off4,off3,off2,off1}
==>  Proc(target,rf,bd.clear(),be',bm,bw,iport,rport,wport,shftr)
      where  target = {pc'[31:28],offset[25:0],2'b00}
             be' = be.enq(BsTemp(pc',
                              Itemp(MWback,True,True,Reg31,pc'+8,-,-)))
```

## R-Type Instructions

```
Rule "Instruction Decode Constant Shifts"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
          if   (op==Special) && canhandle && !STALL(rt)
          &&   bd.notempty() && be.notfull()
       where   BdTemp(pc',inst) = bd.first()
               Mips(op,-,rt,rd,sa,func) = inst
               imm16 = {rt,sa,func}
               canhandle =  (func==Sll) || (func==Sra) || (func==Srl)
==>  Proc(pc,rf,bd.deq(),be',bm,bw,iport,rport,wport,shftr)
       where   be' = be.enq(BsTemp(pc',itemp))
               itemp = (rd==Reg0)?
                         Itemp(MNop,False,False,-,-,-,-):
                         Itemp(mop,True,False,rd,sa,vt,-)
               vt = FORWARD(rt)
               mop = switch(func)
                         case Sll:  MSll
                         case Sra:  MSra
                         case Srl:  MSrl
```

```
Rule "Instruction Decode Triadic"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
         if  (op==Special) && canhandle && !(STALL(rs) || STALL(rt))
         &&  bd.notempty() && be.notfull()
      where  BdTemp(pc',inst) = bd.first()
             Mips(op,rs,rt,rd,-,func) = inst
             canhandle = (func==Add) || (func==Addu)
                         || (func==Sub) || (func==Subu)
                         || (func==And) || (func==Nor)
                         || (func==Or) || (func==Xor)
                         || (func==Slt) || (func==Sltu)
                         || (func==Sllv) || (func==Srav)
                         || (func==Srlv)
==>  Proc(pc,rf,bd.deq(),be',bm,bw,iport,rport,wport,shftr)
      where  be' = be.enq(BsTemp(pc',itemp))
             itemp = (rd==Reg0)?
                       Itemp(MNop,False,False,-,-,-,-):
                       Itemp(mop,True,False,rd,vs,vt,-)
             vs = FORWARD(rs)
             vt = FORWARD(rt)
             mop = switch(func)
                       case Add:  MAdd
                       case Addu:  MAdd
                       case Sub:  MSub
                       case Subu:  MSub
                       case And:  MAnd
                       case Nor:  MNor
                       case Or:  MOr
                       case Xor:  MXor
                       case Slt:  MSlt
                       case Sltu:  MSlt
                       case Sllv:  MSll
                       case Srav:  MSra
                       case Srlv:  MSrl
Rule "Decode Jr"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
         if  !STALL(rs)
         &&  bd.notempty()
      where  BdTemp(pc',inst) = bd.first()
             Mips(Special,rs,-,-,-,Jr) = inst
==>  Proc(vs,rf,bd.clear(),be,bm,bw,iport,rport,wport,shftr)
      where  vs = FORWARD(rs)
```

```
Rule "Instruction Decode Jalr"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
          if   !STALL(rs)
          &&   bd.notempty() && be.notfull()
       where   BdTemp(pc',inst) = bd.first()
               Mips(Special,rs,-,rd,-,Jalr) = inst
==>  Proc(vs,rf,bd.clear(),be',bm,bw,iport,rport,wport,shftr)
       where   be' = be.enq(BsTemp(pc',itemp))
               itemp = (rd==Reg0)?
                          Itemp(MNop,False,False,-,-,-,-):
                          Itemp(MWback,True,True,rd,pc'+8,-,-)
               vs = FORWARD(rs)
```

### B.2.4 Execute Stage Rules

```
Rule "Execute Stage Drop"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
          if   be.notempty()
          &&   ((template==Itemp(MNop,-,-,-,-,-,-)))
       where   BsTemp(pc',template) = be.first()
==>  Proc(pc,rf,bd,be.deq(),bm,bw,iport,rport,wport,shftr)
Rule "Execute Stage Pass"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
          if   ((template==Itemp(MWback,-,-,-,-,-,-)))
          &&   be.notempty() && bm.notfull()
       where   BsTemp(pc',template) = be.first()
==>  Proc(pc,rf,bd,be.deq(),bm',bw,iport,rport,wport,shftr)
       where   bm' = bm.enq(BsTemp(pc',template))
```

```
Rule "Execute 2-to-1 Function"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
         if   canhandle
         &&   be.notempty() && bm.notfull()
      where   BsTemp(pc',template) = be.first()
              Itemp(mop,fwd,-,dest,v1,v2,-) = template
              canhandle = (mop==MAdd) || (mop==MSub)
                          || (mop==MAnd) || (mop==MNor)
                          || (mop==MOr) || (mop==MXor)
                          || (mop==MSlt)
==>  Proc(pc,rf,bd,be.deq(),bm',bw,iport,rport,wport,shftr)
      where   bm' = bm.enq(BsTemp(pc',template'))
              template' = Itemp(MWback,fwd,True,dest,result,-,-)
              result = switch(mop)
                         case MAdd:  v1+v2
                         case MSub:  v1-v2
                         case MAnd:  v1&v2
                         case MNor:  (~(v1|v2))
                         case MOr:  v1|v2
                         case MXor:  v1^v2
                         case MSlt:  (v1[31:31]==v2[31:31])?(v1<v2):v1[31:31]
Rule "Execute Shift Function"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
         if   canhandle
         &&   be.notempty() && bm.notfull()
      where   BsTemp(pc',template) = be.first()
              Itemp(mop,fwd,-,dest,v1,v2,-) = template
              canhandle =  (mop==MSll) || (mop==MSra) || (mop==MSrl)
==>  Proc(pc,rf,bd,be.deq(),bm',bw,iport,rport,wport,shftr)
      where   bm' = bm.enq(BsTemp(pc',template'))
              template' = Itemp(MWback,fwd,True,dest,result,-,-)
              result = shftr.shift(v1[4:0],left,arith,v2)
              left = switch(mop)
                         case MSll:  Left
                         case MSra:  Left
                         case MSrl:  Right
              arith = switch(mop)
                         case MSll:  Logical
                         case MSra:  Arith
                         case MSrl:  Logical
```

```
Rule "Execute Address Calc"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
          if  canhandle
          &&  be.notempty() && bm.notfull()
       where  BsTemp(pc',template) = be.first()
              Itemp(mop,fwd,-,dest,base,offset,v) = template
              canhandle = (mop==MLoad) || (mop==MStore)
==>  Proc(pc,rf,bd,be.deq(),bm',bw,iport,rport,wport,shftr)
       where  bm' = bm.enq(BsTemp(pc',template'))
              template' = Itemp(mop,fwd,False,dest,addr,-,v)
              addr = base+offset
```

### B.2.5 Memory Stage Rules

```
Rule "Memory Stage Pass"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
          if  (template!=Itemp(MLoad,-,-,-,-,-,-))
          &&  (template!=Itemp(MStore,-,-,-,-,-,-))
          &&  bm.notempty() && bw.notfull()
       where  BsTemp(pc',template) = bm.first()
==>  Proc(pc,rf,bd,be,bm.deq(),bw',iport,rport,wport,shftr)
       where  bw' = bw.enq(BsTemp(pc',template))
Rule "Memory Stage Store"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,-,shftr)
          if  bm.notempty()
       where  BsTemp(pc',template) = bm.first()
              Itemp(MStore,-,-,-,addr,-,v) = template
==>  Proc(pc,rf,bd,be,bm.deq(),bw,iport,rport,DportW(addr,v,True),shftr)
Rule "Memory Stage Store Off"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,DportW(-,-,-),shftr)
          if  !(bm.notempty() && Itemp(MStore,-,-,-,-,-,-)==template)
       where  BsTemp(pc',template) = bm.first()
==>  Proc(pc,rf,bd,be,bm,bw,iport,rport,DportW(-,-,False),shftr)
Rule "Memory Stage Load"
     Proc(pc,rf,bd,be,bm,bw,iport,DportR(-,data),wport,shftr)
          if  bm.notempty() && bw.notfull()
       where  BsTemp(pc',template) = bm.first()
              Itemp(MLoad,fwd,-,rd,addr,-,-) = template
==>  Proc(pc,rf,bd,be,bm.deq(),bw',iport,DportR(addr,-),wport,shftr)
       where  bw' = bw.enq(BsTemp(pc',template'))
              template' = Itemp(MWback,fwd,True,rd,data,-,-)
```

### B.2.6 Write-Back Stage Rules

```
Rule "Write-Back Stage"
     Proc(pc,rf,bd,be,bm,bw,iport,rport,wport,shftr)
          if   bw.notempty()
          &&   (template==Itemp(MWback,-,-,-,-,-,-))
       where   BsTemp(pc',template) = bw.first()
               Itemp(MWback,-,-,rd,v,-,-) = template
==>  Proc(pc,rf[rd:=v],bd,be,bm,bw.deq(),iport,rport,wport,shftr)
```

## B.3 Source Term

```
Init Proc(0,-,-,-,-,-,-,-,DportW(-,-,False),-)
```

# Bibliography

[Ann] Annapolis Micro Systems, Inc. A family of reconfigurable computing engines. http://www.annapmicro.com.

[AS99] Arvind and X. Shen. Using term rewriting systems to design and verify processors. *IEEE Micro Special Issue on Modeling and Validation of Microprocessors*, May 1999.

[Aug00] L. Augustsson. BlueSpec language definition. Working Draft, March 2000.

[Ber98] G. Berry. The foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[BRM+99] J. Babb, M. Rinard, C. A. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe. Parallelizing applications into silicon. In *Proceedings of the 7th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'99)*, Napa Valley, CA, April 1999.

[Cad] Cadence Design Systems, Inc. Affirma NC Verilog simulator. http://www.cadence.com/datasheets/affirma_nc_verilog_sim.html.

[Co-] Co-Design Automation, Inc. Superlog. http://www.co-design.com/superlog.

[CW91] R. Camposano and W. Wolf, editors. *High-level VLSI Synthesis*. Kluwer Academic Publishers, 1991.

[DWA+92] D. W. Dobberpuhl, R. T. Witek, R. Allmon, R. Anglin, D. Bertucci, S. Britton, L. Chao, R. A. Conrad, D. E. Dever, B. Gieseke, S. M. N. Hassoun, G. W. Hoeppner, K. Kuchler, M. Ladd, B. M. Leary, L. Madden, E. J. McLellan, D. R. Meyer, J. Montanaro, D. A. Priore, V. Rajagopalan, S. Samudrala, and S. Santhanam. A 200-MHz 64-bit dual-issue CMOS microprocessor. *Digital Technical Journal*, 4(4), 1992.

[Fey99] R. P. Feynman. *The pleasure of finding things out : the best short works of Richard P. Feynman*. Perseus Books, 1999.

[FPF95] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. In *Proceedings of European Design and Test Conference (ED&TC'95)*, Paris, France, March 1995.

[Gal95] D. Galloway. The Transmogrifier C hardware description language and compiler for FPGAs. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'95)*, Napa Valley, CA, April 1995.

[GDWL92] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High-level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992.

[GG97] M. Gokhale and E. Gomersall. High level compilation for fine grained FPGAs. In *Proceedings of the IEEE Symposium on FPGA-based for Custom Computing Machines (FCCM'97)*, Napa Valley, CA, April 1997.

[GM93] M. Gokhale and R. Minnich. FPGA computing in a data parallel C. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'93)*, Napa Valley, CA, April 1993.

[GZD+00] D. D. Gajski, J. Zhu, R. Dömer, A. Gerslauer, and S. Zhao. *SpecC Specification Language and Methodology.* Kluwer Academic Publishers, 2000.

[HHD93] I. J. Huang, B. Holmer, and A. Despain. ASIA: Automatic synthesis of instruction-set architectures. In *Proceedings of the 2nd Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI'93)*, Nara, Japan, October 1993.

[HHD97] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetbility. In *Proceedings of the 34th ACM/IEEE Design Automation Conference (DAC'97)*, Anaheim, CA, June 1997.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2nd edition, 1996.

[Ins88] The Institute of Electrical Electronics Engineers, Inc., New York. *IEEE Standard VHDL Language Reference Manual*, 1988.

[JHA+98] S. P. Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. B, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. *Haskell 98: A Non-strict, Purely Functional Language*, 1998. http://www.haskell.org.

[Joh91] M. Johnson. *Superscalar Microprocessor Design.* Prentice Hall, 1991.

[Kan87] G. Kane. *MIPS R2000 RISC Architecture.* Prentice Hall, 1987.

[Klo92] J. W. Klop. *Term Rewriting System*, volume 2 of *Handbook of Logic in Computer Science.* Oxford University Press, 1992.

[Lis00] M. Lis. Superscalar processors via automatic microarchitecture transformations. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2000.

[LS99] L. Lavagno and E. Sentovich. ECL: A specification environment for system-level design. In *Proceedings of the 36th ACM/IEEE Design Automation Conference (DAC'99)*, New Orleans, LA, June 1999.

[LSI] LSI Logic Corporation. ASIC products. http://www.lsilogic.com/products/asic/index.html.

[LTG97] S. Liao, S. Tjinag, and R. Gupta. An efficient implementation of reactivity for modeling hardware in the Scenic design environment. In *Proceedings of the 34th ACM/IEEE Design Automation Conference (DAC'97)*, Anaheim, CA, June 1997.

[Mar84] P. Marwedel. The Mimola design system: Tools for the design of digital processors. In *Proceedings of the 21st ACM/IEEE Design Automation Conference (DAC'84)*, Albuquerque, New Mexico, 1984.

[Mic99] G. De Micheli. Hardware synthesis from C/C++ models. In *Proceedings of Design, Automation and Test in Europe (DATE'99)*, Munich, Germany, March 1999.

[ML99] J. Matthews and J. Launchbury. Elementary microarchitecture algebra. In *Proceedings of Conference on Computer-Aided Verification*, Trento, Italy, July 1999.

[MLC98] J. Matthews, J. Launchbury, and B. Cook. Microprocessor specification in Hawk. In *Proceedings of the 1998 International Conference on Computer Languages*, Chicago, IL, 1998.

[MLD92] P. Michel, U. Lauther, and P. Duzy, editors. *The Synthesis Approach to Digital System Design.* Kluwer Academic Publishers, 1992.

[MR99] M. Marinescu and M. Rinard. A synthesis algorithm for modular design of pipelined circuits. In *Proceedings of X IFIP International Conference on VLSI (VLSI 99)*, Lisbon, Portugal, November 1999.

[PSH$^+$92] I. Pyo, C. Su, I. Huang, K. Pan, Y. Koh, C. Tsui, H. Chen, G. Cheng, S. Liu, S. Wu, , and A. M. Despain. Application-driven design automation for microprocessor design. In *Proceedings of the 29th ACM/IEEE Design Automation Conference (DAC'92)*, Anaheim, CA, June 1992.

[Raj89] V. K. Raj. DAGAR: An automatic pipelined microarchitecture synthesis system. In *Proceedings of the International Conference on Computer Design (ICCD'89)*, Boston, MA, October 1989.

[Ros00] D. L. Rosenband. Synthesis of multi-cycle operation-centric descriptions. PhD Dissertation Proposal, Massachusetts Institute of Technology, June 2000.

[SAR99a] X. Shen, Arvind, and L. Rudolph. CACHET: An adaptive cache coherence protocol for distributed shared-memory systems. In *Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[SAR99b] X. Shen, Arvind, and L. Rudolph. Commit-reconcile & fences (CRF): A new memory model for architects and compiler writers. In *Proceedings of the 26th International Symposium on Computer Architecture (ISCA'99)*, Atlanta, Georgia, May 1999.

[SM98] L. Séméria and G. De Micheli. SpC: Synthesis of pointers in C, application of pointer analysis to the behavioral synthesis from C. In *Proceedings of International Conference on Computer-Aided Design (ICCAD'98)*, San Jose, CA, November 1998.

[SRI97] SRI International, University of Cambridge. *The HOL System Tutorial, Version 2*, July 1997.

[Sta90] Stanford University. *HardwareC – A Language for Hardware Design*, December 1990.

[Syna] Synopsys, Inc. CBA libraries datasheet. http://www.synopsys.com/products/siarc/cba_lib_one.html.

[Synb] Synopsys, Inc. *HDL Compiler for Verilog Reference Manual.*

[Sync] Synopsys, Inc. *RTL Analyzer Reference Manual.*

[Synd] Synplicity, Inc. *Synplify User's Guide, Version 3.0.*

[TM96] D. E. Thomas and P. R. Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 3rd edition, 1996.

[TPPW99] D. E. Thomas, J. M. Paul, S. N. Peffers, and S. J. Weber. Peer-based multithreaded executable co-specification. In *Proceedings of International Workshop on Hardware/Software Co-Design*, San Diego, CA, May 1999.

[VBR$^+$96] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI*, 4(1), March 1996.

[WC91] R. A. Walker and R. Camposano, editors. *A Survey of High-Level Synthesis Systems.* Kluwer Academic Publishers, 1991.

[Win95] P. J. Windley. Verifying pipelined microprocessors. In *Proceedings of the 1995 IFIP Conference on Hardware Description Languages and their Applications (CHDL'95)*, Tokyo, Japan, 1995.

[Xila] Xilinx, Inc. *The Programmable Logic Data Book.*

[Xilb] Xilinx, Inc. Xilinx foundation series. http://www.xilinx.com/products/found.htm.