

Synthesis of Multi-cycle Circuits from Guarded Atomic Actions

by

Michal Karczmarek

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2011

© Massachusetts Institute of Technology 2011. All rights reserved.

Author
Department of Electrical Engineering and Computer Science
September 1, 2011

Certified by
Arvind
Charles W. and Jennifer C. Johnson Professor
Thesis Supervisor

Accepted by
Leslie Kolodziejcki
Chairman, Department Committee on Graduate Students

Synthesis of Multi-cycle Circuits from Guarded Atomic Actions

by

Michał Karczmarek

Submitted to the Department of Electrical Engineering and Computer Science
on September 1, 2011, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

One solution to the timing closure problem is to perform infrequent operations in more than one clock cycle. Despite the apparent simplicity of the solution statement, it is not easily considered because it requires changes in RTL, which in turn exacerbates the verification problem. Another approach to the problem is to avoid it altogether, by using a high-level design methodology and allow the synthesis tool to generate the design that matches design requirements. This approach hinges on the ability of the tool to be able to generate satisfactory RTL from the high-level description, an ability which often cannot be tested until late in the project. Failure to meet the requirements can result in costly delays as an alternative way of expressing the design intent is sought and experimented with.

We offer a timing closure solution that does not suffer from these problems. We have selected atomic actions as the high-level design methodology. We exploit the fact that semantics of atomic actions are untimed, that is, the time to execute an action does not change its outcome. The current hardware synthesis technique from atomic actions assumes that each action takes one clock cycle to complete its computation. Consequently, the action with the longest combinational path determines the clock cycle of the entire design, often leading to needlessly slow circuits. By augmenting the description of the actions with desired timing information, we allow the designer to split long paths over multiple clock cycles without giving up the semantics of atomicity. We also introduce loops with dynamic bounds into the atomic action description. These loops are not unrolled for synthesis, but the guards are evaluated for each iteration. Our synthesis results show that the clock speed and performance of circuits can be improved substantially with our technique, without having to substantially change the design.

Thesis Supervisor: Arvind

Title: Charles W. and Jennifer C. Johnson Professor

Acknowledgments

This thesis will be completed nearly 11 years after I first arrived at MIT. I have met and parted with countless friends, and had many great experiences, many of which were enabled by the wonderful community at MIT.

Arvind has been a wonderful adviser, especially given our differences in personalities. He has been nearly infinitely patient as I searched for a good method to present this work, and much earlier, when my the efforts to come up with an efficient synthesis method for asynchronous circuits were not bearing fruit. His help and guidance have been invaluable, and I will be forever grateful for his faith in me eventually finishing my thesis.

I also must thank my committee, Krste Asanović and Steve Ward, who have also patiently waited for me to finish, and accommodated my graduation deadline from California and Maine respectively.

Muralidaran Vijayaraghavan has been of invaluable help in the latter stages of work on this thesis, while I was working in California, and after I've come back to Boston. He has helped me come up with clean explanations of the synthesis scheme and with discussions of some low-level details when I most needed them.

Michael Pellauer and Nirav Dave have been of immense help whenever I had questions about or problems with the Bluespec compiler. Without them I would never have gained enough insight into the inner workings of the current synthesis technique to be able to improve on it significantly.

My wife, Anne, has put up with the uncertainty of whether I will graduate and the imminent "I have to work on my thesis" for far longer than she should have. She has been my pillar of moral support and my greatest cheerleader. My parents have also always been supportive of my finishing, even when it was not clear that I had a path forward.

This research has been supported by National Science Foundation (grant #CCF-0541164 "Complex Digital Systems") and Nokia.

Contents

1	Introduction	17
1.1	The Case for Multi-cycle Atomic Action Design	17
1.1.1	Summary of Contributions	20
1.2	Prior Work	21
1.3	Thesis Organization	25
2	Bluespec and Semantics of Multi-cycle Guarded Atomic Action Machines	27
2.1	BMC: The Essence of Bluespec	27
2.2	BMC Example: GCD	28
2.3	Standard Bluespec Flat Model Circuit	30
2.4	Standard Hierarchical Bluespec Interfaces	31
2.5	Multi-cycle Syntax	34
2.6	Multi-cycle Operational Semantics	36
2.7	Preview of Guarded Atomic Action Scheduling Concepts	38
2.8	Rule Lifting Procedure	39
3	Flat, Fixed, Multi-Cycle Rule Scheduling	41
3.1	Two-clock-cycle GCD	41
3.2	A Reference Scheduler for Multi-cycle Rules	44
3.3	Read-Write Policy for Concurrent Scheduling of Rules	45
3.4	Opportunistic-Commit Scheduler	46
3.5	Greedy-Reservation Opportunistic-Commit (GROC) Scheduler	47

3.6	Greedy Rsv Scheduler and Cmt Scheduler for the GROC Scheduler	49
3.7	Scheduling Invariants	50
3.8	Scheduling Algorithm	54
3.9	Correctness of GROC Scheduler	56
3.10	The GROC Scheduler for GCD	56
4	Flat, Flexible, Multi-Cycle Scheduling	59
4.1	Loop-based GCD Example	59
4.2	(Valid, Changed, Data) 3-tuples	60
4.3	Architectural Registers	62
4.4	Syntax-directed Translation of Multi-cycle Expressions	63
4.4.1	Expression Interface	63
4.4.2	Register, Constant and Variable	63
4.4.3	Let Expression	64
4.4.4	Primitive Operator	65
4.4.5	Conditional Expression	65
4.4.6	Delay Expression	66
4.4.7	While Expression	67
4.5	Syntax-directed Translation of Multi-cycle Actions	68
4.5.1	Action Interface	68
4.5.2	Register Assignment	69
4.5.3	Conditional Action	70
4.5.4	Parallel Action	70
4.5.5	Let Action	71
4.6	Syntax-directed Translation of Rules	71
4.6.1	Rule Interface	71
4.6.2	Rule Circuit	71
4.6.3	Combining Rules for Register Writes	72
4.7	Module Model	73
4.7.1	Single-cycle Equivalency	73

5	Flat Scheduling In Presence of Guards	75
5.1	Speculative Scheduler	75
5.2	Speculative Expressions	78
5.2.1	Speculative Expression Interface	78
5.2.2	Register and Constant Expressions	79
5.2.3	Speculative Let Expression	79
5.2.4	Primitive Operator Expression	79
5.2.5	Speculative Conditional Expression	81
5.2.6	Speculative Guarded (When) Expression	81
5.2.7	Delay Expression	81
5.2.8	Loop Expression	83
5.3	Speculative Actions	84
5.3.1	Speculative Action Interface	84
5.3.2	Speculative Let Action	84
5.3.3	Speculative Register Assignment Action	85
5.3.4	Speculative Conditional Action	85
5.3.5	Speculative Guarded (When) Action	85
5.3.6	Speculative Parallel Action Composition	87
5.4	Speculative Rules	87
5.5	Speculative Module Model	88
6	Hierarchical Multi-cycle Synthesis	89
6.1	Reservation Required	89
6.2	Hierarchical Speculative Scheduling	91
6.3	Hierarchical Speculative Method Interfaces	94
6.4	Speculative Method Call Protocol	95
6.5	Method Port Sharing and Multiplexing	98
6.6	Hierarchical Expressions	104
6.6.1	Hierarchical Expression Interface	104
6.6.2	Hierarchical Value Expressions	104

6.6.3	Hierarchical Let Expression	105
6.6.4	Hierarchical Primitive Operator Expression	105
6.6.5	Hierarchical Conditional Expressions	107
6.6.6	Hierarchical Guarded (When) Expression	107
6.6.7	Hierarchical Delay Expression	107
6.6.8	Value Method Call Expression	109
6.6.9	Loop Expression	109
6.7	Hierarchical Actions	111
6.7.1	Hierarchical Action Interface	111
6.7.2	Hierarchical Let Action	112
6.7.3	Hierarchical Register Assignment Action	112
6.7.4	Hierarchical Conditional Action	112
6.7.5	Hierarchical Guarded (When) Action	114
6.7.6	Hierarchical Parallel Action Composition	114
6.7.7	Action Method Call	114
6.8	Module Integration	116
6.9	Summary of Scheduling Restrictions	116
7	Results and Analysis	119
7.1	Modulo in GCD	119
7.2	CPU with Division	122
7.3	FFT	124
7.4	Viterbi	128
7.5	Analysis	130
7.5.1	Area Overhead due to Added Flip-Flops	131
7.5.2	Long Combinational Paths in Control Logic	132
8	Conclusion and Future Work	135
8.1	Conclusion	135
8.2	Future Work	137
8.2.1	Pipelining	137

8.2.2	Modular Scheduling	138
8.2.3	Multi-cycle Control Signals	138
8.2.4	Combine with Ephemeral History Registers	138
8.2.5	Synthesis of Asynchronous Logic	139

List of Figures

2-1	Single-clock-cycle BMC Language	28
2-2	GCD example	29
2-3	GCD circuit with single clock cycle <i>GCD</i> rule	29
2-4	Standard Bluespec Model. Thin wires represent boolean values, while thick wires represent data of any format.	30
2-5	Single Clock Cycle Call Protocol Transition Diagram	33
2-6	GCD Parent Module	34
2-7	Multi-cycle Language	35
2-8	Multi-cycle Operational Semantics	37
3-1	GCD example with a 2 clock cycle <i>GCD</i> rule	42
3-2	2 clock cycle GCD circuit	43
3-3	One Rule-At-a-Time Multi-cycle model	44
3-4	Opportunistic Scheduler	47
3-5	The GROC Fixed Delay Scheduler. The green box contains the essence of the GROC scheduler.	48
3-6	GROC Scheduler Invariants	50
3-7	Sequentially Composable Rules	51
3-8	Rule Stretching (Delayed Commit)	52
3-9	Livelock avoidance	53
3-10	RsvScheduler Procedure	54
3-11	CmtScheduler Procedure	55
3-12	Complete GCD Example	57

3-13	Complete GCD Control and Scheduling Equations	57
4-1	Loop-based GCD example	60
4-2	Loop-based GCD Circuit	61
4-3	Architectural Register	63
4-4	Expression Interface	63
4-5	Register, Constant and Variable	64
4-6	Let Expression	64
4-7	Primitive Operator	65
4-8	Conditional Expression	65
4-9	Delay Register	66
4-10	Loop Expression	67
4-11	Action Interface	69
4-12	Register Write Action	69
4-13	Conditional Action	70
4-14	Parallel Action	70
4-15	Let Action	71
4-16	Rule Interface	72
4-17	Rule Circuit	72
4-18	Register Enable Circuit	73
4-19	Full Module with GROC Scheduler. The essence of the GROC scheduler is indicated by the green box.	74
5-1	Fully Speculative Scheduler. The modified essence of the GROC scheduler is indicated by the green box.	78
5-2	Speculative Expression Interface	78
5-3	Speculative Value Read Expressions	79
5-4	Speculative Let Expression	80
5-5	Speculative Primitive Operation Expression	80
5-6	Speculative Conditional Expression	81
5-7	Speculative Guarded Expression	82

5-8	Speculative Delay Expression: $\text{delay}(e)$	82
5-9	Speculative Loop Expression. We use nR as shortcut for notRdy . . .	83
5-10	Speculative Action Interface	84
5-11	Speculative Let Action	84
5-12	Speculative Register Assignment Action	85
5-13	Speculative Conditional Action	86
5-14	Speculative Guarded Action	86
5-15	Speculative Parallel Actions	87
5-16	Speculative Rule Circuit	87
5-17	Speculative Rule Circuit	88
6-1	Fully Speculative Scheduler. Note: $RR_A = NO$, $RR_B = YES$. The modified essence of the GROC scheduler is indicated by the green box.	93
6-2	Multi-cycle Method Interfaces	95
6-3	Multi-cycle Speculative Method Call Protocol	97
6-4	One-Hot MUX with 3-tuple selectors	100
6-5	3-tuple MUX	101
6-6	Argument MUX	102
6-7	Commit MUX	103
6-8	Hierarchical Expression Interface	104
6-9	Hierarchical Value Read Expressions	105
6-10	Hierarchical Let Expression	106
6-11	Hierarchical Primitive Operation Expression	106
6-12	Hierarchical Conditional Expression	107
6-13	Hierarchical Guarded Expression	108
6-14	Hierarchical Delay Expression	108
6-15	Value Method Call Expression: $m.f(e)$	109
6-16	Hierarchical Loop Expression	110
6-17	Hierarchical Action Interface	111
6-18	Hierarchical Let Action	112

6-19	Hierarchical Register Assignment Action	113
6-20	Hierarchical Conditional Action	113
6-21	Hierarchical Guarded Action	114
6-22	Hierarchical Parallel Actions	115
6-23	Hierarchical Action Method Call	115
6-24	Value method caller and callee composition	117
6-25	Action method caller and callee composition	118
7-1	Different Remainder Versions	120
7-2	Multi-cycle GCD results. Using loop-based remainder significantly reduces the area.	121
7-3	Simple 4-stage pipelined processor	122
7-4	Partial Listing of ALU implementation. The DIV instruction takes 17 clock cycles to complete, while other instructions take only 1 clock cycle. The non-strict <code>if</code> operator selects the correct result and returns it as soon as it is available.	124
7-5	CPU Synthesis Results	124
7-6	FFT Design	126
7-7	FFT Synthesis Results. *Pareto points. **3.53ns was obtained by synthesizing with a test harness. 1.17ns is anomalous and was obtained by synthesizing without a test harness. Likely cause of the anomaly is that without a testing harness the circuit has exactly one register and no register-to-register path.	127
7-8	Viterbi Algorithm	130
7-9	Viterbi Synthesis Results. FS means forward step. TB means trace-back. *Pareto points.	131
7-10	Adding <code>delay</code> to condition of <code>if</code>	134

Chapter 1

Introduction

1.1 The Case for Multi-cycle Atomic Action Design

As the technology for manufacturing silicon chips allows for ever larger designs, the complexity of new designs increases, and with it the effort put into building them. A designer must worry about many problems: the design must perform well, use little power and fit in a small area. Above all, the design must perform its function correctly when presented with valid inputs. Assessing components of the design for these requirements is difficult enough, but nothing less than simultaneously satisfying all of them is tolerated. Design of a modern microprocessor takes many years from first concept and specification to first silicon. Only the largest companies with products which will sell in many millions of units can afford this kind of effort.

Most designs today are expressed in RTL, in a language like Verilog or VHDL. Some custom designs are synthesized using high-level synthesis, to reduce the effort necessary to produce the design. High-level synthesis allows for a description of the solution to be presented in a way that designers find more expressive and natural. The most common form of high-level synthesis is C to RTL synthesis. The premise of this form of synthesis is that writing software is widely recognized as a much easier task than writing RTL. If a design can be synthesized from a software description, then the final result will be arrived at much faster and the correctness will be transferred automatically from the simpler software description to the more complicated RTL

description.

The problem with this premise is that designing RTL requires many tradeoffs to meet the three basic goals beyond correctness. High-level synthesis from C does not allow for making such trade-offs easily. The designer is given knobs, but not real low-level control of the final RTL. Design exploration can change from expressing the design in a different way, to playing with these knobs. Sometimes this will lead to a design meeting specifications, but sometimes it will not.

Another approach to high-level synthesis is to use a language that is more amenable to hardware design. Much as hardware designers specialize their designs for particular purpose, language designers specialize their languages for particular use and domain. One such language is Bluespec [12]. Bluespec is designed around the idea of guarded atomic actions: a program is described as a combination of state and atomic actions. Guards are assigned to atomic actions to prevent chaotic functionality and guard against actions performing invalid updates. An atomic action can fire and update the state at any time, provided its guard evaluates to true.

The atomicity of actions and updates provides a certain comfort to the designer. The designer can think about one action at a time. The abstraction of atomic actions breaks away from the always-update-everything approach of low-level RTL design. The compiler will ensure that execution of conflicting actions does not overlap. The compiler also provides feedback on which actions are and are not composable, making thinking about composability of updates simpler.

Bluespec language has no concept of clock cycles. As such, Bluespec compiler is free to generate rules that take multiple clock cycles. For simplicity reasons, the current Bluespec compiler only generates actions that complete in a single clock cycle. This provides predictability to the designer: the final clock speed is dependent only on how complex the atomic actions are. Similarly, the size and power of the design is also in the hands of the designer.

Bluespec also lends itself well to various forms of design exploration. From simple modifications such as changing the size of an ROB [17] to more complex exploration such as many approaches to folding an FFT computation block [19], Bluespec makes

it easy to substitute different implementations of parts of a design without having to change the entire design.

The low-level control available to the designer in Bluespec can however prove itself to be an Achilles’s heel. If the clock speed of a design is found to be unsatisfactory, that action can be split into two. This is the approach taken while folding an FFT design. The problem (low clock speed) is solved by abandoning the language-provided comfort of atomic actions. Now a single computation or update must use multiple actions to complete. The designer again has to worry about the concurrency of the design. The concern can be isolated to a single design block (ex. FFT) but the peace of mind provided by atomic action abstraction is diminished.

The approach of action splitting or other slow-path optimizations in hardware design are similar in some ways to other specialization present in various computer design fields. For instance, parallel programming requires intense optimization of usually a small portion of the program. The remainder of the design takes very little time thus it can, and often does, run on a single thread. The effort of optimization is best spent on improving “hot” code. In hardware design, the effort often goes to optimizing the “hot” path as well. The FFT optimization effort is an example of that.

Unfortunately, hardware design suffers from a complication that does not exist in software design. In software, code that is used only 1% of the time has only a 1% impact on performance of the overall design. In hardware, a slow path that is used only 1% of the time can cause the clock of the entire design to be slow at all times. Thus hardware designers cannot simply disregard the marginal portions of their design and concentrate on the hot paths. The entire design must be carefully balanced to provide good performance on all paths. There are many situations where designers would like to insert a slow path in a design to simplify it (ex: exception handling), but they must build pipelines or other structures to maintain good performance for the entire chip. In Bluespec, even very rarely used actions must be split if they are too slow for the overall design.

In this thesis we present solutions that completely solve this timing problem in Bluespec. Instead of splitting slow actions we allow the designer to build multi-cycle

actions which provide predictable improvements in clock cycle time. This allows the designer to concentrate on expressing the design with the comfort of guarded atomic actions. The designer retains very close control over the synthesized results. Flip-flops are inserted only where requested, but can be easily inserted on any slow path. Our compiler guarantees atomic completion of computation and updates.

The RTL produced by our compiler is not pipelined. That is, there is only one instance of each action active during any clock cycle. We leave automatic pipelining of designs for future work. Our techniques do allow for solving real timing closure problems, while making the solutions much easier to understand the implement than prior state of the art. Furthermore, designers can utilize our techniques to gauge the impact pipelining would have on the performance of their designs.

We also introduce new semantics to Bluespec. Our if-expressions only wait for the used result, so rarely used paths can take many clock cycles without penalty to hot paths. We also introduce dynamic loops which are not automatically unrolled. Our loops are simple and effective, leaving all control in the hands of the designer without having to rely on awkward knobs for optimization, as is often the case in synthesis from C. These improvements increase expressiveness of the language and the design, and are natural to designers used to high-level languages.

Finally, we provide interfaces for Bluespec methods which are inherently delay-agnostic. This allows for simple substitution of different implementations of IP blocks without concern about the time taken internally to complete a computation. In today’s market, where companies differentiate different chips in their product lines by the performance or count of individual components, having an ability to plug in differently performing IP blocks without modifying the remainder of the design can prove very useful.

1.1.1 Summary of Contributions

This thesis makes the following contributions:

- Extension of Bluespec language to include multi-cycle concepts without chang-

ing atomic action semantics.

- Improvement of power of Bluespec language by expressing combinational loops that do not unroll.
- New syntax-directed compiling procedure which subsumes the current single-cycle state of the art without compromising quality.¹
- Modification of hierarchical synthesis interfaces, which must change to adapt to multi-cycle actions. Our new interfaces are such that it should be possible to generate wrappers for the old interfaces to include legacy IP blocks in our designs.

1.2 Prior Work

This thesis is largely based on the Bluespec and guarded atomic actions work done in the past. James Hoe developed the efficient scheduling techniques for synthesis of flat modules [29, 28]. The basic design of the scheduler in this thesis draws from the Esposito scheduler [23]. Rosenband developed a modular, hierarchical scheduling method, together with method interfaces, both of which were a starting point for our modular synthesis [34].

While this thesis is concerned with synthesizing guarded atomic actions to take multiple clock cycles, Rosenband has proposed the opposite approach of allowing multiple not-necessarily sequentially composable actions to complete in a single clock cycle through the use of Ephemeral History Registers [6, 7].

Dave et al., have introduced a concept of a sequential composition within an atomic action, which can be used to describe multi-cycle designs [18]. Their approach is far more powerful than our delay operator, and full implementation in hardware requires introduction of shadow state. Aside from that, the operational semantics of their BTRS is largely identical to our BMC.

¹We do not address or consider the problem of `RWire` construct. Our intuition suggests that the straight-forward implementation of `RWire` is relatively simple, though its semantics are complicated by multi-cycle nature of our scheme.

There are many commercial and free tools for high-level synthesis. One of the most-popular approaches is to extend and/or restrict C or C++ to allow expression of various desired or undesired constructs. Common restrictions include disallowing pointers, virtual functions, etc. Common extensions include explicit parallelism constructs and channel-based blocking communication constructs. With many different features from different approaches, lack of standardization impedes wide-adoption.

Using C or C++ without any language extensions can help in creation of a first draft of a design. Standard software development tools can be used to write and debug the first draft. Once the design is bug-free, a high level synthesis tool takes over and allows for design exploration and synthesis.

Perhaps the first attempt at synthesis of C to RTL was the Cones project by Stroud et al [40]. Cones was capable of only synthesizing combinatorial portion of the design and the memories had to be explicitly defined. As such, it only needed to work with a subset of C. The results of the synthesis were used for production chips from AT&T. The designers using Cones found that they were able to create their designs in half the time, compared to then standard techniques.

One of the early unmodified C to RTL compiling schemes that attempted to include more advanced features of C was SpC[35]. SpC allowed use of pointers to arrays. The implementation used SUIF[44], and the compiler applied pointer analysis to replace pointer loads and stores with references to appropriate arrays. Curiously, SpC controlled the timing of the computation by incrementing a `clk` variable.

A more automated approach to C synthesis was taken by Babb et al[9]. Their compiler concentrates on partitioning the program data into small memories, which are accessed by independent FSMs. FSMs communicate with each other through virtual wires.

Another early successful method for synthesis from C was used in C2Verilog [38, 39]. In this system the designer explores the design through selecting synthesis parameters. One of the new optimization techniques was to give the compiler the ability to share logic between different parts of the design, if synthesis parameters required it.

DEFACTO was another early system in which the compiler iterates over different solutions, searching for a balanced design[37].

CASH [14] converts the input program into hyperblocks, which are synthesized separately. Each hyperblock is converted into a data-flow graph. The generated circuits can be synchronous or asynchronous, as the communication between hyperblocks uses the Two-Phase Bundled Data protocol commonly used in asynchronous hardware. This approach is notable, as this thesis was born from a desire to synthesize asynchronous circuits from guarded atomic actions.

Today, major EDA vendors offer tools for synthesis of unmodified C to RTL with various knobs for meeting design criteria [16, 1, 4].

When it comes to synthesis from C-like languages, it is important to note that for the most part they are C-like in their syntax only. These languages were created to make synthesis easier and give the designer more control over the synthesized results. They usually add very non-C-like features, such as explicit parallelism, message passing through some sort of channels and timing semantics. These are domain-specific language specializations, which render the code thoroughly incompatible with ANSI C. Many of these features have appeared in other languages, such as Esterel[13]. In fact, Esterel has been used for high-level synthesis before [11, 31, 22].

From the approaches that significantly extended C for synthesis, Handel-C [2] is notable for this thesis, because both it and we use a delay operator to indicate a clock cycle delay. Assignments in Handel-C also require a clock cycle to propagate, just as register writes in Bluespec. Communication between statements is done through assignment to variables or through blocking channels. Availability of data on the channels becomes a scheduling constraint, similarly to data flow. There is an explicit parallel construct and loops, but no pointers. Since Handel-C does not have a concept of atomic actions, it does not require the same scheduling effort that Bluespec does.

HardwareC was an early attempt at a C-derived language for hardware synthesis [30]. HardwareC uses concepts of ports and input/output communication, message passing, various degrees of parallelism. HardwareC was used in the Olympus synthesis system [21] which was a complete synthesis suite including tools to constraint-driven

synthesis, user-driven synthesis, synthesis feedback and technology mapping tools.

Transmogrieff C [25] was another C-derived language, targeted at FPGAs. It added the concept of ports and parallelism. The timing model was simple and predictable: loops and function calls start new clock cycles, all other logic is combinational.

SpecC is another HDL derived from C [3]. It adds the concepts of events for synchronization, signals and channels for communication, behaviors, interfaces and ports for building containers for computation, buffers for storage, and time. There are ways to explicitly build pipelines, FSMs, and concurrent computation.

SystemC [41] is a C++ library that enables event-driven simulation of concurrent processes. A subset of SystemC constructs is synthesizable. SystemC enjoys wide industry support.

SystemVerilog [5] brings Verilog features to a C++-like syntax and is used for verification, replacing a more traditional C++/Verilog mix. Again, a subset of SystemVerilog is synthesizable, but adoption for synthesis is not as wide as for verification.

The GARP scheduler [15] uses techniques borrowed from software pipelining to generate efficient schedules for a CPU attached to a rapidly reconfigurable coprocessor. The scheduler creates hyperblocks which are executed on the programmable array, but it excludes paths which do not show promise for this optimization because they are exercised too infrequently, and would not provide a net benefit considering the cost of reconfiguring the programmable array. Those paths are executed on the slower CPU. Our approach is similar, in that we enable isolation of infrequent paths and executing them over many clock cycles, to focus designer's energy on frequently used paths.

There are also many projects which used a more specialized approach. For instance, early on, Cathedral II project [32] targeted synthesis for DSP algorithms. Cathedral II used Silage for their HDL [27]. Being rooted in DSP domain, Silage operated on infinite data streams. A more modern approach for DSP synthesis is to start with MATLAB, which is commonly used for prototyping in the DSP domain [10, 8].

Our scheduling approach has some similarities to the scoreboard scheduling technique, originally used in the CDC 6600 [42], or more modern processors with branch mispredicts. The traditional scoreboard deals with a dynamic stream of instructions which share physical resources and which have some sequentiality constraints based on data dependencies. The rules with their atomicity and resource constraints can be thought of as a fixed set of instructions that need to be scheduled (dispatched) each clock cycle because the execution of a rule potentially affects the availability of rules for scheduling in the next cycle.

Transactional Memories [26, 36] deal with atomic updates and are conceptually close to the problems discussed in this thesis. However, the current TM systems assume that the read set and write set (the domain and range) of a transaction cannot be statically estimated by a compiler. Thus, the TM systems deal with a much more difficult problem which requires shadow state and difficult computation involving intersection and union of read sets and write sets. However, our work can be viewed as an implementation for small, statically analyzed transactions.

Our work also has some similarities to automatic pipelining [33]. Pipelining concentrates on creating a bypass network such that the pipeline can consume new data as often as possible, without having to wait for the previous data to be fully processed. This thesis proposes to split computation across multiple clock cycles, but not to pipeline it. As such we do not generate a bypass network or any pipeline control logic. We leave pipelining for future work.

This thesis began with an idea of synthesizing asynchronous circuits from guarded atomic actions. The association of a valid wire with a data value and the complex port handshake were partially inspired by handshaking methods learned from asynchronous circuits [20].

1.3 Thesis Organization

In Chapter 2, we begin with the discussion of the current state of synthesis from guarded atomic actions. We continue with description of BMC, our source language,

including operational semantics. In Chapter 3 we present the basic principles of multi-cycle scheduling in flat modules with fixed computational delays. We update our synthesis scheme to allow dynamic computational delays, including loops, in Chapter 4. Chapter 5 introduces speculative scheduling which helps with fairness. Chapter 6 extends our synthesis to hierarchical modules. We discuss synthesis results and provide some analysis in Chapter 7. Our conclusion and future work are presented in Chapter 8.

Chapter 2

Bluespec and Semantics of Multi-cycle Guarded Atomic Action Machines

In this chapter we will present some essential Bluespec and multi-cycle concepts. We will begin by introducing single-cycle BMC, a language which we use to describe guarded atomic-action machines. We will illustrate the principles of how such guarded atomic-action machines function with an example using GCD. We will present the general structure of circuits generated by the current Bluespec compiler. Finally, we will introduce multi-cycle BMC and present its operational semantics.

2.1 BMC: The Essence of Bluespec

Figure 2-1 presents BMC, a language used to describe guarded atomic action machines. This language was described by Dave et al[18].

In this language, every machine consists of multiple modules, arranged in a tree hierarchy. Every module holds some registers which represent the state of the machine. Modules contain rules which represent guarded atomic actions. Each rule consists of exactly one action and a guard. Modules also contain value methods and action methods, which are used to communicate with the module. A value method computes

```

m ::= Module name
      Register r v
      [ t = e ]
      [ Rule R a when e ]
      [ ActMeth G a when e ]
      [ ValMeth F e when e ]

```

```

a ::= a | a
      || r := e
      || if e then a

```

```

e ::= r || c || t
      || op (e, e)
      || (t = e in e)
      || if e then e else e

```

Figure 2-1: Single-clock-cycle BMC Language

a result based on module state and returns it as a result. An action method executes an action that is triggered using an external interface. Only the direct parent of a child module can call its methods. Ultimately all method calls are initiated by a rule.

Actions consist of register assignments, conditional actions, parallel composition of actions and action method calls. Expressions consist of reading a value (register, constant or variable), basic operators on expressions, let expressions and guarded expressions.

2.2 BMC Example: GCD

Consider the example in Figure 2-2, which describes a module to compute the GCD of two numbers using the Euclidean method. The module has two registers *x* and *y*, one rule called *GCD*, and two interface methods called *Seed* and *Result*. The *Seed* method stores the initial numbers in registers *x* and *y*, the *GCD* rule computes successive remainders of the inputs until the remainder becomes 0, and the *Result* method returns the computed answer. The rule has the guard *when y* \neq 0 which

```

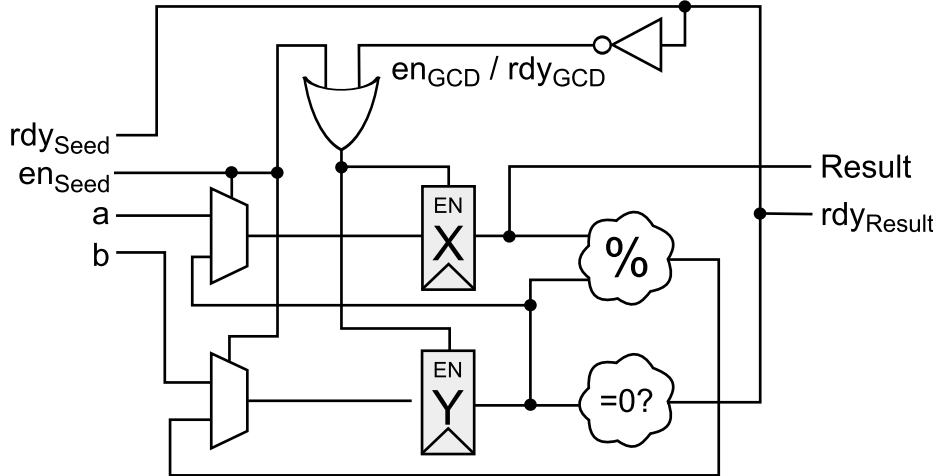
Module doGcd
  Register  $x = 0, y = 0$ 
  Rule GCD ::  $(x := y \mid y := x \% y)$  when  $y \neq 0$ 
  ActMeth Seed( $a, b$ ) ::  $(x := a \mid y := b)$  when  $y = 0$ 
  ValMeth Result :: return  $x$  when  $y = 0$ 

```

Note: % operator represents remainder computation

Figure 2-2: GCD example

must be *true* for the rule to execute. Similarly, the methods have guards which must be *true* before the method can be applied. The guards for the *Seed* and *Result* methods indicate that they can be activated only when the module is not already busy computing, (i.e. *when* $y = 0$). “%” represents the remainder operator while “|” represents parallel composition of two actions and is a commutative operator. Thus, if we write $(x := y \mid y := x)$ the values of x and y would be swapped at the end of the action. Also parallel assignments to a state variable in an action are illegal. The source language allows conditional assignments to a register but provides no ability to sequence actions in a rule.



Note: $start_{GCD}$ and rdy_{GCD} hold the same value. We have abbreviated *enable* with just *en*.

Figure 2-3: GCD circuit with single clock cycle *GCD* rule

In Figure 2-3 we provide a circuit implementation of the *GCD* example. The x and y registers are updated when either the *GCD* rule is active or the *Seed* method is started. In this example, the conditions for these two actions are mutually exclusive because of the guards. The compiler must make sure that it never asserts the *enable* wire for a rule unless its *rdy* signal is asserted. If *Seed*'s *enable* signal is asserted, the input values a and b are copied into x and y , respectively. When *Result*'s guard (rdy_{Result}) is *true*, the user can read the output, which is simply the value of register x . Since *Result* does not modify any of the registers in the module, the user can read the output without further communication. Rule *GCD* is enabled whenever its guard evaluates to *true*. When *GCD* is executed, it will copy y to x , and write the remainder, $x \% y$, to y . *GCD* rule completes in a single clock cycle, but computing the final result requires firing it many times.

2.3 Standard Bluespec Flat Model Circuit

In previous section we presented a simple BMC module and a circuit it produces. Since the module is very simple, compiling it to a circuit was also fairly simple. More complicated modules require a rigorous methodology for compiling.

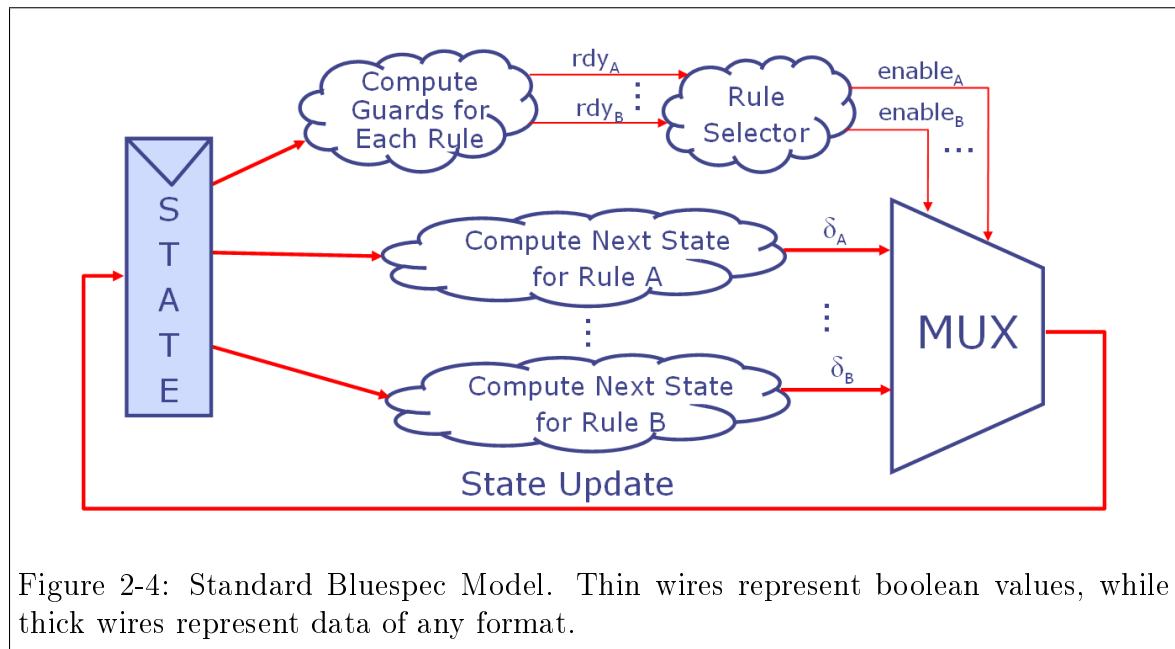


Figure 2-4 presents the Bluespec flat module template circuit. Every clock cycle

the state (defined in registers) is read, and passed to guard expressions and action circuits. The guards for all rules evaluate independently, and produce *rdy* signals. These signals are all inspected by a **Rule Selector** (scheduler) which selects a non-conflicting set of ready rules to be executed on that particular clock cycle. This set is signaled using the *enable* signals. (A set of rules is non conflicting if there exists an ordering of all rules in the set which does not produce a read-after-write ordering of data.) The state update MUX uses the updates produced by all the rules and the start signals to produce updates to machine state. The next clock cycle the whole process begins from scratch with reading of the now updated state.

In order to make the process of selecting rules for execution efficient, the scheduler circuit imposes a total ordering of the rules to express their priorities. Rules with higher priority are considered earlier for execution. We call this ordering an urgency list or urgency ordering.

The template described here is used for compiling flat modules, that is modules which expose no methods and contain no child modules. We refer to this as the Hoe-Arvind scheduler. [29]

2.4 Standard Hierarchical Bluespec Interfaces

In the previous section we presented the template for synthesizing circuits for flat Bluespec modules. In this section we will present the approach to hierarchical synthesis described in [34]. We call this the Rosenband-Arvind scheduler.

A quick glance at the grammar in Figure 2-1 tells us that it is easy to write a program in which multiple rules call a single method in a particular child module. However, physically, we can only generate circuits for a limited number of instances of such a method. For our purposes we will assume that each method will have exactly one instance of its circuit present. As a result, methods are now considered to be scarce resources, and we must manage them carefully. Any two rules which require access to the same method are now considered to be conflicting, with some exceptions. This extends our definition of conflicting rules from section 2.3.

Each module independently decides which rules it should activate in a particular clock cycle independently of other modules. The only communication between the schedulers is through *rdy* and *enable* signals for method calls. Each scheduler selects a non-conflicting set of rules to activate, taking into account the possibility of conflicts through method calls.

The prior modular synthesis method has been described in [34]. Just as prior flat module synthesis, this prior method assumes that all rules complete their computation in a single clock cycle.

The basic approach is to define a standard interface for methods to communicate with their callers. Modules with methods expose an instance of this interface for every method. The interfaces facilitate coordination of execution between the caller module and the child module, and passing data between them.

The interfaces of methods consist of the following wires:

- *rdy* – this is the result of evaluating the method guard. *rdy* is an output from the method.
- *enable* – asserting enable indicates that the method is being called in that clock cycle. At the end of the clock cycle, the action method will commit its updates to the architectural state if its *enable* wire is asserted. Only action methods have *enable* wires.
- *parameter* – any method may accept inputs from the caller. Those inputs must be valid in the clock cycle in which the method is called.
- *result* – value methods must return the computed value in the result wire.

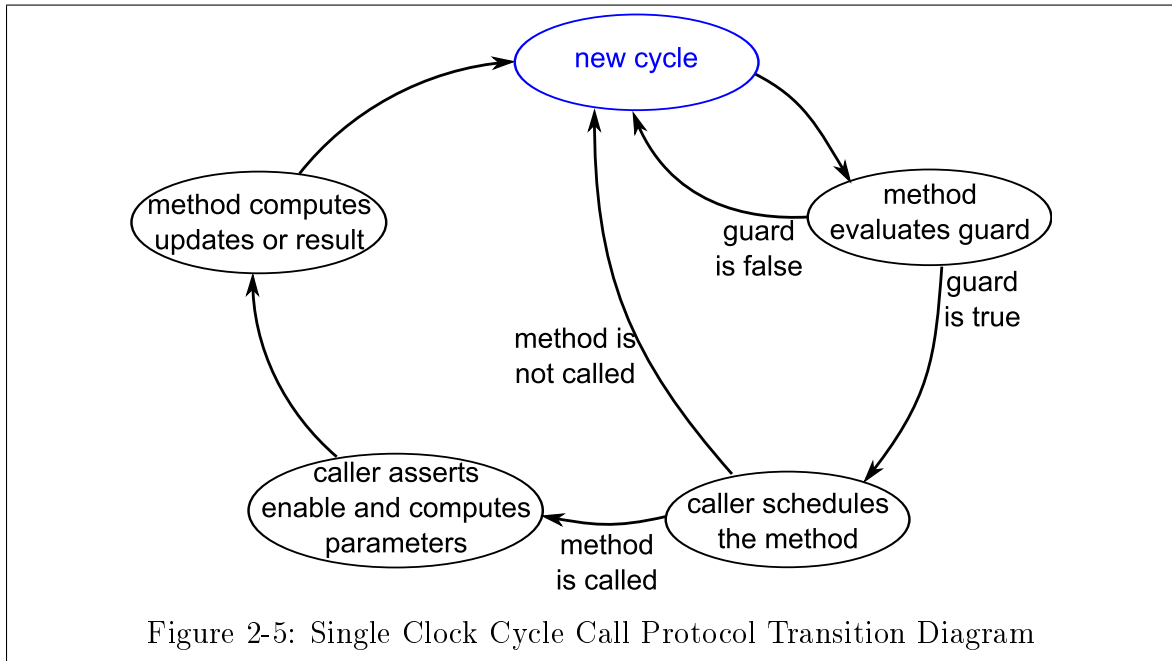
The method call protocol is listed below:

1. The method evaluates its guard, and outputs it to *rdy*. If the guard evaluates to false, the method is unavailable for calling in this clock cycle.
2. If the method’s guard evaluates to true, the parent module considers the method for calling. This requires scheduling all the rules and methods for this clock

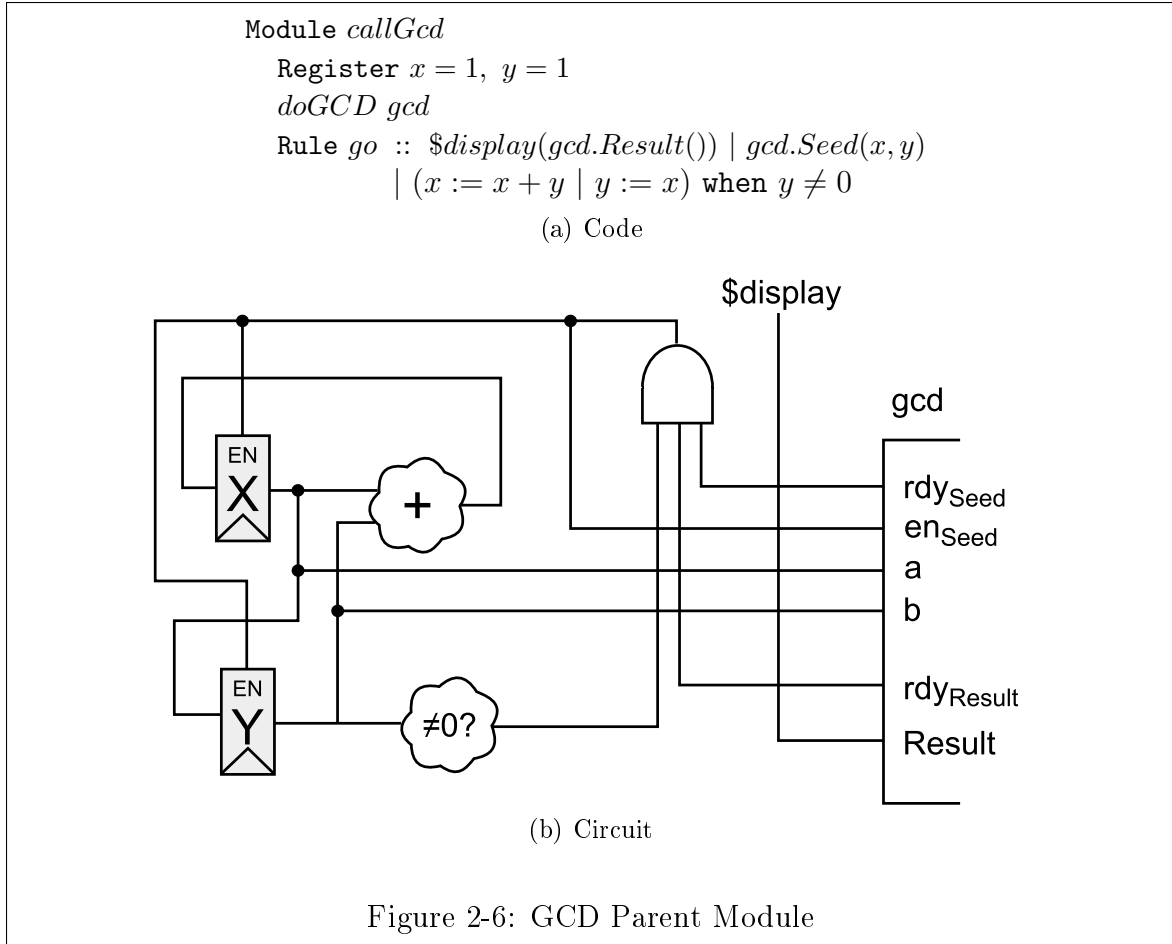
cycle, and evaluating any predicates in actions or expressions calling the method. If a method is called, the *enable* signal is asserted.

3. The parent module evaluates any parameters to the method and asserts their values on the *parameter* wire.
4. (Value method) The method evaluates its result and eventually puts it on the *result* wire
5. (Action method) The method evaluates its updates and updates the architectural state at the end of the clock cycle.

The transition diagram of the single clock cycle call protocol can be found in Figure 2-5.



The example in Figures 2-2 and 2-3 describes a *doGCD* module with one rule and two methods. In Figure 2-6 we present a parent module which calls the methods in *doGCD*.



2.5 Multi-cycle Syntax

Figure 2-7 presents the syntax of our multi-cycle language. The language is quite similar to that in Figure 2-1. Notice that for the most part, the syntax of the language does not deal with the concept of computation over multiple clock cycles. The only sign of multi-cycle computation is the presence of the **delay** expression operator. The delay expression operator functionally acts as an identity, but indicates to the compiler that a clock cycle delay is requested on the expression path.

We have also created a separate expression type, flat expressions. The only difference between regular expressions and flat expressions is that flat expressions cannot call value methods. This is necessary because our circuits cannot handle value method calls from inside loops.

This thesis will present four schemes for compiling to multi-cycle circuits, each

$m ::=$	<code>Module name</code>	// Module definition
	<code>[Register $r := v$]</code>	// Register with initial value
	<code>[Rule $R a$ when e]</code>	// Rule (3, 4)
	<code>[Module – type m]</code>	// Child module (6)
	<code>[Rule $R a$]</code>	// Rule (5, 6)
	<code>[ActMeth $G a$]</code>	// Action method (6)
	<code>[ValMeth $F e$]</code>	// Value method (6)

(a) Module Syntax

$a ::=$	<code>$r := e$</code>	// Register assignment
	<code> $a \mid a$</code>	// parallel action composition
	<code> if e then a</code>	// conditional action
	<code> $t = e$ in a</code>	// let action
	<code> a when e</code>	// when action (5, 6)
	<code> $m.g(e)$</code>	// action method call (6)

(b) Action Syntax

$e ::=$	<code>$r \mid t \mid c$</code>	// register, variable read, constant
	<code> op (e, e)</code>	// primitive operator
	<code> $t = e$ in e</code>	// let expression
	<code> if e then e else e</code>	// conditional expression
	<code> e when e</code>	// when expression (5, 6)
	<code> delay (e)</code>	// delay expression
	<code> $m.f(e)$</code>	// value method call (6)
	<code> (while e_l do [$t = e_l$] return t)</code>	// while loop expression (4,5, 6)

(c) Expression Syntax

$e_f ::=$	<code>$r \mid t \mid c$</code>	// register, variable read, constant
	<code> op (e_f, e_f)</code>	// primitive operator
	<code> $t = e_f$ in e_f</code>	// let expression
	<code> if e_f then e_f else e_f</code>	// conditional expression
	<code> delay (e_f)</code>	// delay expression
	<code> (while e_f do [$t = e_f$] return t)</code>	// while loop expression (4,5, 6)

(d) Flat Expression Syntax

Figure 2-7: Multi-cycle Language

scheme with the ability to compile more language features. The syntax in Figure 2-7 includes all the language features we can compile with our most advanced scheme. We have annotated the features that can only be compiled with more advanced compilation schemes using the chapter in which those features will be supported. Chapter

3 will serve as an introduction to multi-cycle compilation. The scheme presented in Chapter 3 will use a rigid counter-based scheduler. Loops will not be an available language feature. Method calls also will not be available, thus making the compilation scheme flat. Chapter 4 will build on Chapter 3 by using a more flexible data validity tracking mechanism, and introducing loops. Chapter 5 will introduce a speculative scheduling approach and allow each action and expression their own individual guards. Finally 6 will introduce hierarchical compilation, allowing communication between modules through method calls. We will present results, analysis and future work in Chapter 7, and conclusions in Chapter 8.

2.6 Multi-cycle Operational Semantics

Figure 2-8 presents the operational semantics for the syntax presented in Section 2.5. The triplet $\langle S, U, B \rangle$ represents the current register state, planned updates and local variable bindings. NR represents the “not-ready” value, which can be stored in the bindings but cannot be assigned to a register. If we cannot find a rule in operational semantics to proceed forward, the encompassing action can not be completed. For example, we cannot pass NR to a method, and trying to do so prevents the method call from happening.

In our language method’s guards are implicitly obeyed by the caller. That is if the guard of a method evaluates to *false*, then the method cannot return a value or perform its updates and the encompassing expression or action do not have operational semantics for such cases.

The feature distinguishing our language from that of [18] is the **delay** operator. The **delay** operator is only present in expressions. Functionally, the **delay** operator behaves as an identity. In practice we will convert each use of a **delay** operator to introduce a one clock cycle delay in computation of the expression, thus giving the programmer control over multi-cycle computations. Our conjecture is that the **delay** operator and an identity are functionally the same. The simple argument for this to be true is that the atomic action model executes one action at a time, and all

reg-update	$\frac{\langle S, U, B \rangle \vdash e \rightarrow v, v \neq \text{NR}}{\langle S, U, B \rangle \vdash r := e \rightarrow \{\}[v/r]}$
if-true	$\frac{\langle S, U, B \rangle \vdash e \rightarrow \text{true}, \langle S, U, B \rangle \vdash a \rightarrow U'}{\langle S, U, B \rangle \vdash \text{if } e \text{ then } a \rightarrow U'}$
if-false	$\frac{\langle S, U, B \rangle \vdash e \rightarrow \text{false}}{\langle S, U, B \rangle \vdash \text{if } e \text{ then } a \rightarrow \{\}}$
a-when-true	$\frac{\langle S, U, B \rangle \vdash e \rightarrow \text{true}, \langle S, U, B \rangle \vdash a \rightarrow U'}{\langle S, U, B \rangle \vdash a \text{ when } e \rightarrow U'}$
par	$\frac{\langle S, U, B \rangle \vdash a_1 \rightarrow U_1, \langle S, U, B \rangle \vdash a_2 \rightarrow U_2}{\langle S, U, B \rangle \vdash a_1 \mid a_2 \rightarrow (U_1 \uplus U_2)}$
a-let-sub	$\frac{\langle S, U, B \rangle \vdash e \rightarrow v, \langle S, U, B[v/t] \rangle \vdash a \rightarrow U'}{\langle S, U, B \rangle \vdash t = e \text{ in } a \rightarrow U'}$
a-meth-call	$\frac{\langle S, U, B \rangle \vdash e \rightarrow v, v \neq \text{NR}, m.g = \langle \lambda t.a \rangle, \langle S, U, B[v/t] \rangle \vdash a \rightarrow U'}{\langle S, U, B \rangle \vdash m.g(e) \rightarrow U'}$

reg-read	$\langle S, U, B \rangle \vdash r \rightarrow S(r)$
const	$\langle S, U, B \rangle \vdash c \rightarrow \underline{c}$
variable	$\langle S, U, B \rangle \vdash t \rightarrow B(t)$
op	$\frac{\langle S, U, B \rangle \vdash e_1 \rightarrow v_1, v_1 \neq \text{NR}, \langle S, U, B \rangle \vdash e_2 \rightarrow v_2, v_2 \neq \text{NR}}{\langle S, U, B \rangle \vdash e_1 \text{ op } e_2 \rightarrow v_1 \text{ op } v_2}$
tri-true	$\frac{\langle S, U, B \rangle \vdash e_1 \rightarrow \text{true}, \langle S, U, B \rangle \vdash e_2 \rightarrow v}{\langle S, U, B \rangle \vdash e_1 ? e_2 : e_3 \rightarrow v}$
tri-false	$\frac{\langle S, U, B \rangle \vdash e_1 \rightarrow \text{false}, \langle S, U, B \rangle \vdash e_3 \rightarrow v}{\langle S, U, B \rangle \vdash e_1 ? e_2 : e_3 \rightarrow v}$
e-let-sub	$\frac{\langle S, U, B \rangle \vdash e_1 \rightarrow v_1, \langle S, U, B[v/t] \rangle \vdash e_2 \rightarrow v_2}{\langle S, U, B \rangle \vdash t = e_1 \text{ in } e_2 \rightarrow v_2}$
e-meth-call	$\frac{\langle S, U, B \rangle \vdash e \rightarrow v, v \neq \text{NR}, m.f = \langle \lambda t.e_b \rangle, \langle S, U, B[v/t] \rangle \vdash e_b \rightarrow v'}{\langle S, U, B \rangle \vdash m.f(e) \rightarrow v'}$
delay-expr	$\frac{\langle S, U, B \rangle \vdash e \rightarrow v}{\langle S, U, B \rangle \vdash \text{delay}(e) \rightarrow v}$
e-while-loop-done	$\frac{\langle S, U, B \rangle \vdash e_1 \rightarrow \text{false}, \langle S, U, B \rangle \vdash t \rightarrow B(t)}{\langle S, U, B \rangle \vdash \text{while } e_1 \text{ do } [t = e_2] \text{ return } t \rightarrow B(t)}$
e-while-loop-next	$\frac{\langle S, U, B \rangle \vdash e_1 \rightarrow \text{true}, \langle S, U, B \rangle \vdash e_2 \rightarrow v_1, \langle S, U, B[v_1/t] \rangle \vdash \text{while } e_1 \text{ do } [t = e_2] \text{ return } t \rightarrow v_2}{\langle S, U, B \rangle \vdash \text{while } e_1 \text{ do } [t = e_2] \text{ return } t \rightarrow v_2}$

Figure 2-8: Multi-cycle Operational Semantics

register updates happen at the end of action execution. Thus slowing computation down must not change the behavior of the design. This feature is essential to ensure that the semantics of BMC is the same as semantics of Bluespec.

Since we have not yet specified the method of implementing the `delay` operator, we are free to experiment. One valid implementation is to insert flip-flops for all instances of `delay` operators, thus forcing the paths from expression inputs to final outputs to explicitly take multiple clock cycles. Another valid implementation would be to create multi-cycle combinational paths. Such a solution would prevent pipelining of the design but save on power and area. We could also implement a hybrid approach.

Also, since the `delay` operator is local to the expression which contains it, the value held by any flip-flops generated to hold its state not explicitly accessible to the designer. In fact, the `delay` operator may be introduced by a tool asked to improve clock cycle time on a design with slow paths.

2.7 Preview of Guarded Atomic Action Scheduling Concepts

This thesis presents a method for scheduling multi-cycle guarded atomic action designs. In each successive chapter we will add new features to our scheduling technique: multi-cycle scheduling with fixed computation times; scheduling with loops; scheduling with embedded guards; and finally modular scheduling.

It is important to consider what it means to schedule a rule in a circuit. Since circuits are physical devices, we are really concerned about the observable side-effects of scheduling the rule. We also notice that at a high level, circuits can be thought of as always on: all registers are read on every clock cycle and all data is always computed. The updates for all actions are always being computed. So scheduling an action is really about finding a good clock cycle to commit the updates that have already been computed, rather than finding a good clock cycle to compute the updates.

We will use two different methods for keeping track of when the updates are fully

computed. The first method, used in Chapter 3, will simply count the number of clock cycles from the last update to the domain of an action. The method used in later chapters will keep track of validity of data using two extra signals: *valid* and *changed*. The *valid* signal will tell us that a particular data wire holds data that has been fully propagated from the current architectural state. The *changed* wire will tell us that the architectural registers used to compute the data on a particular wire have just changed. We will use the *changed* wire to reset the value of the *valid* wire after an update. We need these two wires because our scheme does not propagate data as tokens, but rather waits for the multi-cycle circuits between the domain and range of a rule to fully propagate the data. This allows us to perform incremental updates to our computation. In either case, if we used the approach of treating data as tokens, we would have to have a method to kill tokens when they are invalidated by an update. This would require a mechanism similar to our *changed* wire.

The exception to the reasoning about actions always computing their updates is when multiple actions share a scarce resource. If that is the case, we must pick which action should have access to such a resource, to the detriment of other actions. This is the case when we introduce modularity, and rules and methods may share scarce method ports. This scarcity of method ports will become particularly troublesome when a scarce port is needed to evaluate a guard. We will need a mechanism for giving the access to the port to the guard.

Throughout this thesis, we generally use term “cycle” to mean “clock cycle”. Whenever we refer to multi-cycle, we always mean multiple clock cycles. Single-cycle always means single clock cycle. We always use “iteration” to refer to a “loop cycle”.

2.8 Rule Lifting Procedure

Our circuits will utilize a central scheduler, unlike those described in [34], . That is, only the top-level module will contain a scheduler. To accomplish this, we need to make sure that only the top-level module contains any rules. We will utilize the following procedure to accomplish this:

Procedure RuleLift:

1. For each module m in design, traversed in post-order:
 - 1.1 If m is top-level module, finish
 - 1.2 For every rule r in m :
 - 1.2.1 Convert **Rule** r a into **ActMeth** g_r a
 - 1.2.2 Add the following rule r' to parent of m : **Rule** r' : $m.g_r()$

Chapter 3

Flat, Fixed, Multi-Cycle Rule Scheduling

In this chapter we will present a technique for scheduling multi-cycle rules in a flat module. We will begin with a motivational example based on the GCD design from the previous chapter. We will then present a simple reference scheduler design. We will follow that with an exploration of atomic consistency in presence of multi-cycle rules, and finish with a formal algorithm for building circuits which correctly implement an efficient multi-cycle rule scheduler. We will end this chapter with a simple example scheduler.

This chapter assumes that the number of clock cycles required to complete a particular rule is known statically, i.e., known at compilation time. This assumption is enforced by not accepting the loop construct of the language and by always assuming worst-case scenario propagation delays. We also assume that rule guards are single-cycle, that is they are not allowed to contain the `delay` operator.

3.1 Two-clock-cycle GCD

Consider the GCD design shown in Figure 3-1. The design is identical to the design from Figure 2-2, except that the modulo computation is performed by a $\%_2$ function, indicating that the result takes two clock cycles to produce the final result. The new

```

Module doGcd2
  Register  $x = 0, y = 0, t$ 
  Rule GCD ::  $(x := y \mid y = \%_2(x, y))$  when  $y \neq 0$ 
  ActMeth Seed( $a, b$ ) ::  $(x := a \mid y := b)$  when  $y = 0$ 
  ValMeth Result :: return  $x$  when  $y = 0$ 

```

Figure 3-1: GCD example with a 2 clock cycle *GCD* rule

GCD rule is now a multi-cycle rule. Assuming that the remainder operation was the critical path in our circuit, the new implementation may improve the clock cycle time by up to a factor of 2. This cycle time reduction does not come for free. We have introduced extra state, and will have to address the problem of maintaining rule atomicity. Producing the final result still requires many firings of the *GCD* rule.

Figure 3-2 provides a circuit implementation of the code from Figure 3-1. There are two notable differences between this circuit and one in Figure 2-3. The first difference is that we have split the computation of the remainder into two clock cycles (represented by the clock waveform below $\%_2$ operator circuit). The second is that we have changed the signal indicating the completion of the *GCD* rule. The new signal is called *commit_{GCD}* and is asserted one clock cycle after the rule becomes active because the computation of the remainder now takes 2 clock cycles.

Assuming that the implementation of $\%_2$ is efficient, we may have reduced the clock cycle of this module by as much as half. Of course this clock cycle reduction is unlikely to speed up the computation of GCD itself because GCD computation will require twice as many clock cycles as before. However, it should speed up the encompassing design if computing the remainder in GCD was the critical path of the whole design. Note that this example is exceedingly simple, because it only deals with scheduling a single rule. Real designs require scheduling multiple rules.

Also note, that the specifics of how $\%_2$ circuit is generated are not important here. $\%_2$ can decompose the remainder computation into two parts and insert a register to store an intermediate result. Alternatively $\%_2$ can be a 2 clock cycle combinational circuit. All that matters for correctness here is that after two clock cycles of stable

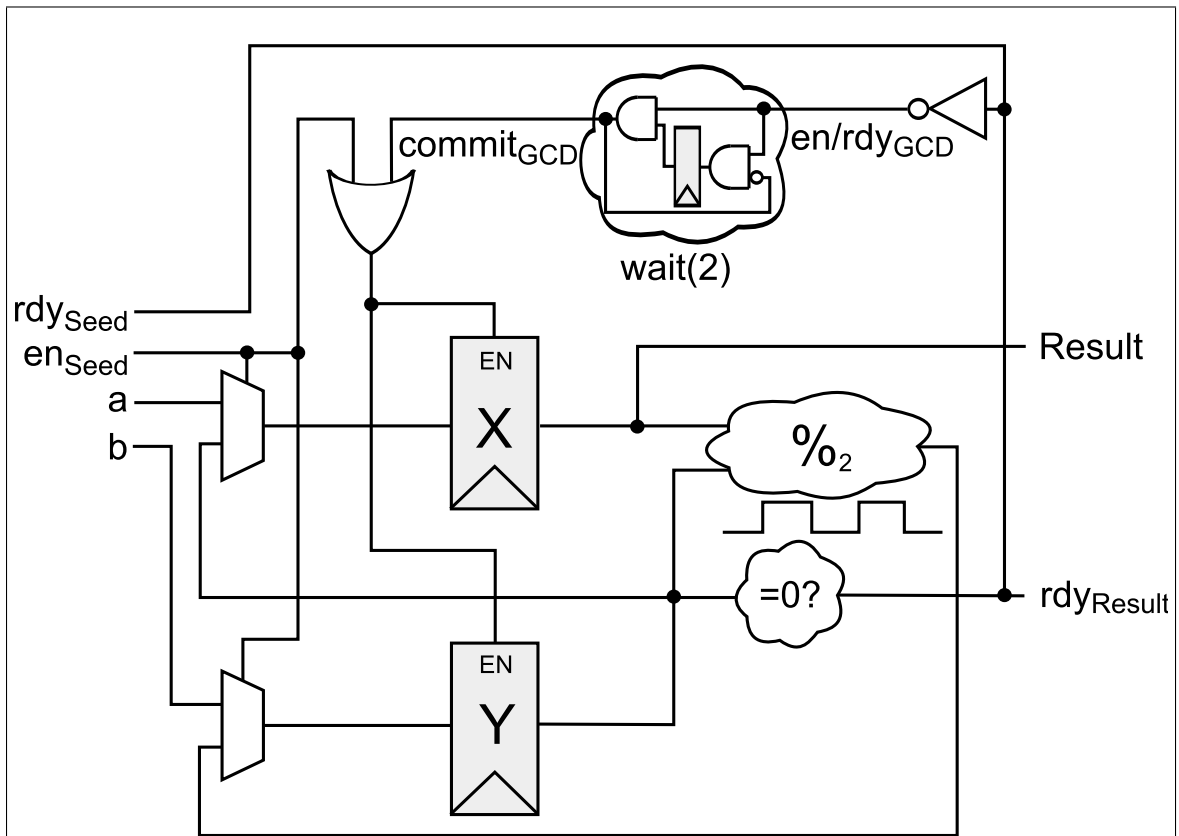
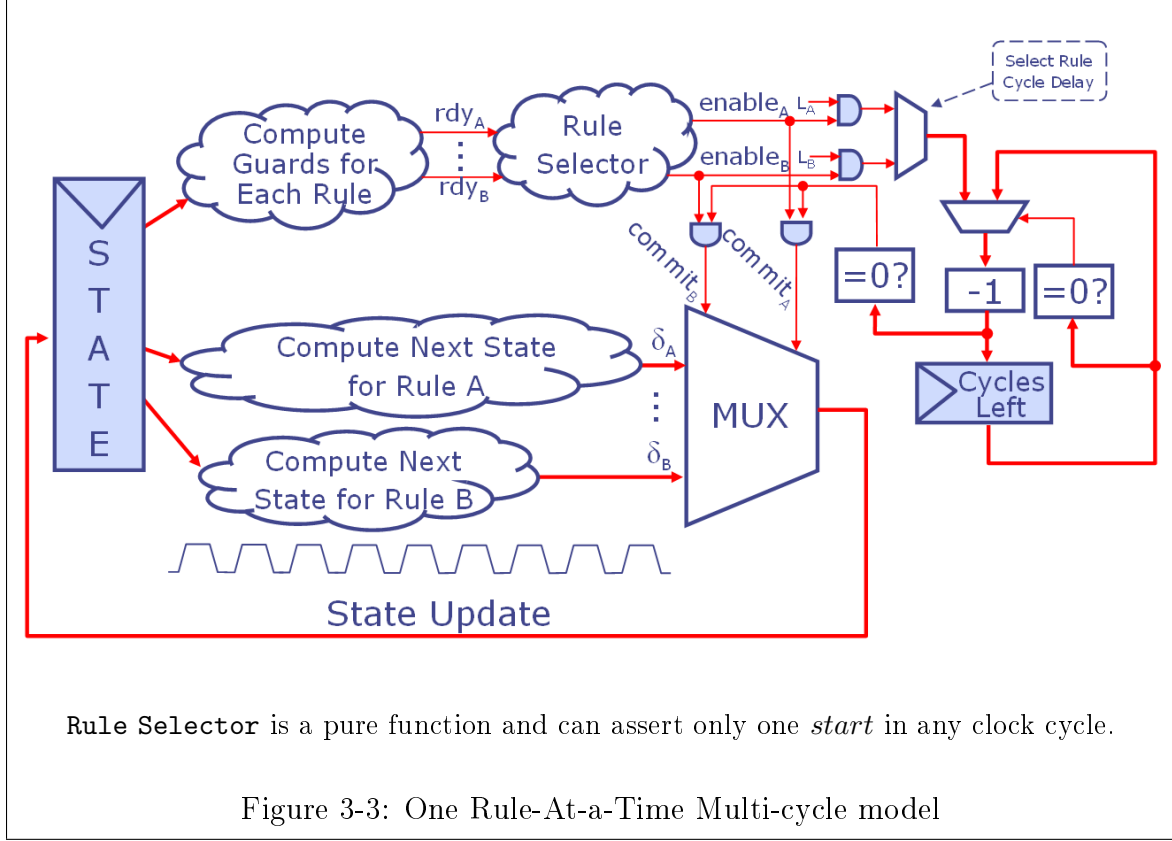


Figure 3-2: 2 clock cycle GCD circuit



inputs to $\%_2$, the output should be fully propagated and stable.

3.2 A Reference Scheduler for Multi-cycle Rules

Consider a reference schedule for multi-cycle execution of rules, shown in Figure 3-3. The reference scheduler allows only a single rule to be active in any clock cycle, which is selected by the **Rule Selector**. The fundamental difference between the models in Figures 2-4 and 3-3 comes from separating the *enable* and *commit* signals and having a counter (**CyclesLeft**) which counts down as the data propagates through the currently active rule. When all rules take one clock cycle, as in Figure 2-4, the *enable* and *commit* signals are one and the same. In our new model, asserting *enable_A* indicates that rule *A* is firing and *commit_A* indicates that rule *A* is committing its updates. Once *enable_A* is asserted it remains active until *commit_A* is asserted.

The first clock cycle in which *enable_A* is asserted, **CyclesLeft** is set to L_A , the computation delay of rule *A*. When **CyclesLeft** reaches 0, the *commit_A* signal for

rule A is asserted and the computed data is stored in the target state registers. The `CyclesLeft` counter is loaded with a new value only when its present value is zero, and is decremented by 1 otherwise. Thus, the effect of the scheduler is felt only on those clock cycles when a scheduled rule has just completed or when the whole system is idling. This implementation guarantees rule atomicity because inputs to an active rule cannot change as no other rule is active.

The circuits to compute the next state can be built using multi-cycle combinational paths or using flip-flops for temporary variables. These flip-flops are not globally visible. All guards are still evaluated in a single clock cycle.

3.3 Read-Write Policy for Concurrent Scheduling of Rules

When multiple multi-cycle rules are active concurrently, we have to make sure that an active rule cannot destroy the input registers of another active rule. The procedure to guarantee this condition has to know exactly when a rule reads and writes its various registers. For example, we could assume that a rule reads all its input registers in the first clock cycle it becomes active and after that those input registers can be changed without affecting the behavior of the rule. This will potentially introduce a lot of “shadow state”. On the writing side, one can imagine writing the various output registers of a rule in any clock cycle whenever the associated data become available. These read-write policies can get complicated and their effect on the final results can be quite difficult to analyze. Therefore, we use a simple *read-throughout* – *write-last* policy.

Read throughout policy means that a rule can read its input registers in any clock cycle the rule is active, and consequently, the register must hold its value during that time. *Write-last* policy means that a rule does not commit any new values to its output registers until the very last clock cycle in which the rule is active. This read-write policy simplifies the scheduling problem and helps implementing if-statements

correctly.

In order to define the interaction between rules in reading and writing each others input and output registers we need the following definitions for rules A and B :

Definition (Range and Domain): Range of rule A is the set of all registers A can update and is written as $R[A]$. Domain of rule A is the set of all registers which A can read potentially and is written as $D[A]$.

Definition (Conflict Free Rules): Two rules A and B are Conflict Free ($A \text{ CF } B$) iff $(D[A] \cap R[B] = \emptyset) \wedge (D[B] \cap R[A] = \emptyset) \wedge (R[A] \cap R[B] = \emptyset)$.

Definition (Sequentially Composible Rules): Two rules A and B are Sequentially Composible ($A < B$) iff $(R[A] \cap (D[B] \cup R[B]) = \emptyset) \wedge (D[A] \cap R[B] \neq \emptyset)$.

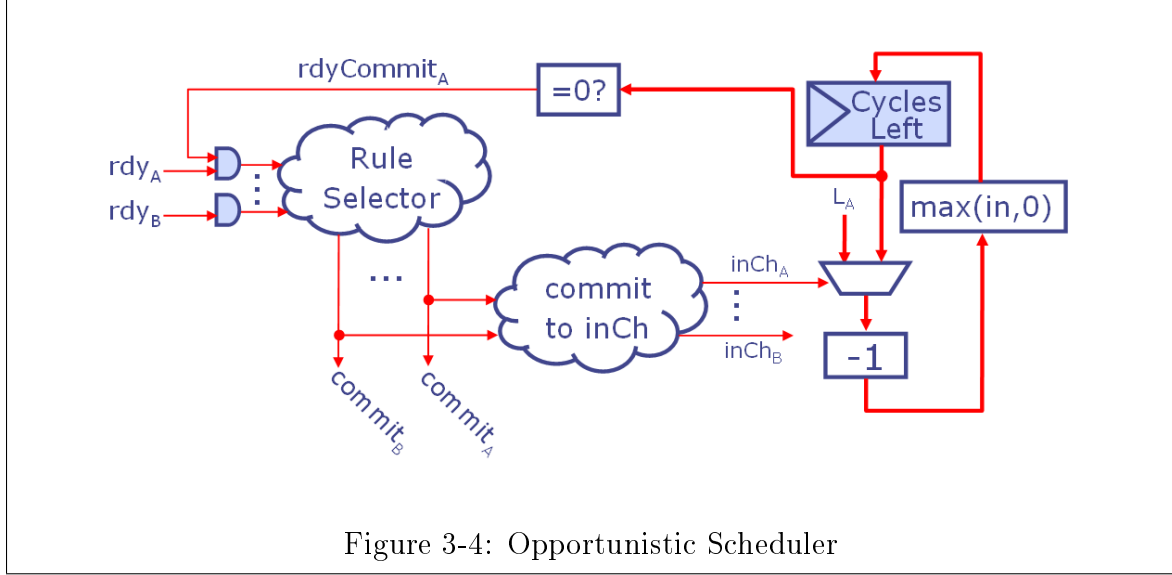
Note: this definition is slightly different from [29] by precluding Conflict Free rules from being treated as Sequentially Composible Rules.

Definition (Conflicting Rules): Two rules A and B are Conflicting ($A <> B$) iff $\neg((A \text{ CF } B) \vee (A < B) \vee (B < A))$.

We will use these definitions to decide which rules can be scheduled to execute concurrently.

3.4 Opportunistic-Commit Scheduler

In Figure 3-4 we present an improvement over the scheduler in Figure 3-3. Notice that the circuits in the scheduler in Figure 3-3 compute updates for all the rules all the time even though they commit at most one rule in a clock cycle. The new scheduler opportunistically commits a non-conflicting subset of rules among those rules whose guards are true, and who have completed their next state computations. In order to keep track of which rules are ready to commit we have created a separate **CyclesLeft** counter for each rule. The counter for rule A is reset whenever a commit affects any of the registers that A reads. In order to determine when some register in $R[A]$ has changed, we watch all *commit* commands and compute all the *inCh* (input change) signals as follows:



$$inCh_A(t) = \bigvee_{B \in \mathcal{R}} ((A <> B \vee A < B) \wedge commit_B(t))$$

$$CyclesLeft_A(t) = inCh_A(t-1) ? (L_A - 1) : \max(cycLeft_A(t-1) - 1, 0)$$

$$rdyCommit_A(t) = (cycLeft_A(t) = 0)$$

where L_A is the propagation delay of rule A , which is the maximum number of clock cycles it takes for data from $D[A]$ to propagate through A 's computational logic.

One interesting feature of the scheduler in Figure 3-4 is that it can use the same the **Rule Selector** circuit to generate *commit* signals that is used in the current Bluespec (BSV) compiler to generate *enable* signals. The **Rule Selector** requires no new static analysis because we have removed the rules which are not ready to commit from the set of rules for consideration for scheduling; among these rules the Hoe-Arvind scheduler [29] is guaranteed to pick only those rules for committing that will not destroy the rule atomicity property.

3.5 Greedy-Reservation Opportunistic-Commit (GROC) Scheduler

The opportunistic scheduler in previous section suffers from the problem of unfairness. A rule that completes its computation quickly will be ready to commit early. A slow rule which depends on the output of a fast rule will never complete its computation,

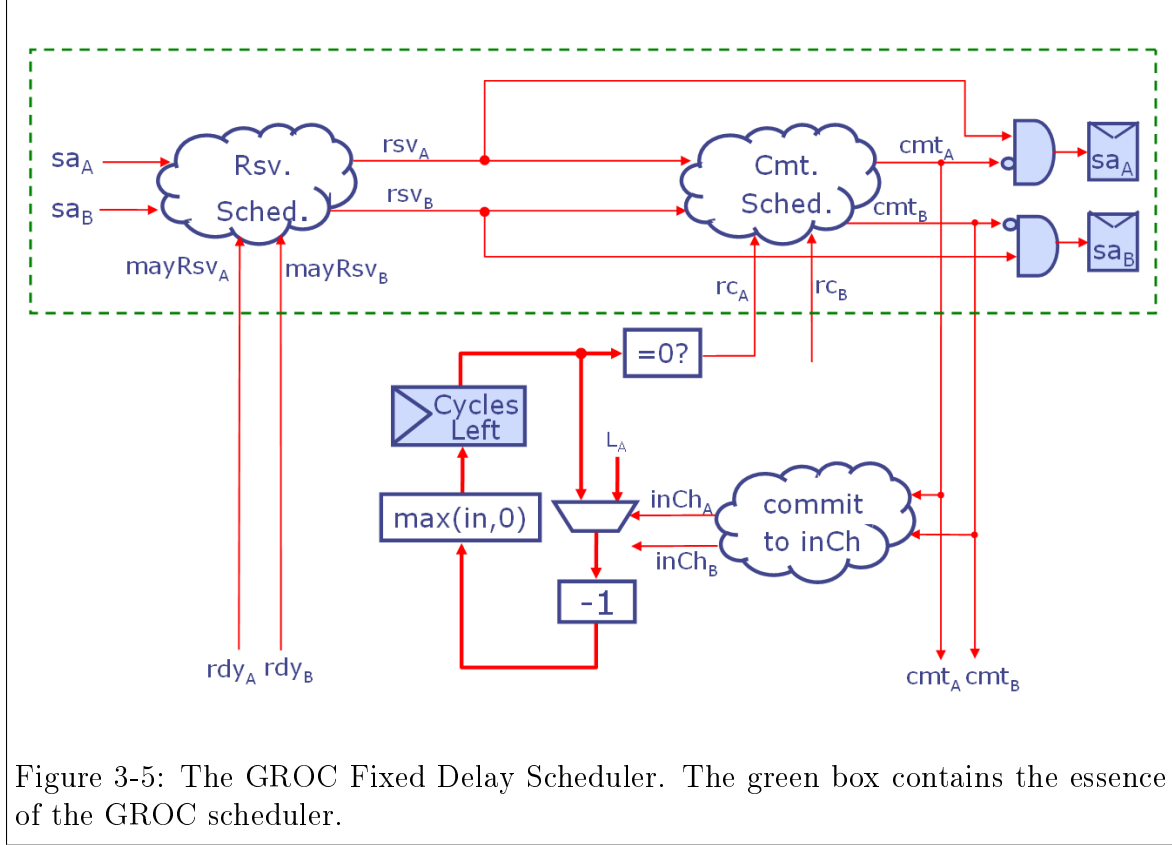


Figure 3-5: The GROC Fixed Delay Scheduler. The green box contains the essence of the GROC scheduler.

because its **CyclesLeft** counter will continually be reset before it counts down to 0.

In order to overcome the problem of unfairness we will use a system where the scheduler uses a *rsv* (reserve) command to reserve a turn for a rule to commit its updates. Once a *rsv* command has been issued, the rule is considered active and only an active rule is allowed to commit its update. The scheduler will continue issuing the *rsv* command for the rule as long as the rule is active. In order to keep track of which rules are active from clock cycle to clock cycle we need to introduce an extra bit of state per rule, sa_A (still active A). It represents whether the particular rule was active in the previous clock cycle and did not commit, thus is still active in the current clock cycle.

$$sa_A(t) = rsv_A(t-1) \wedge \neg cmt_A(t-1)$$

$$sa_A(0) = false$$

Once activated, the only way to deactivate a rule is by issuing its *cmt* command. The ability to reserve a commit slot in the future ensures that no rule can be blocked

permanently from committing by another conflicting rule. The new scheduler is shown in Figure 3-5. It uses the same definitions for $inCh_A$, $CyclesLeft_A$ and $rdyCommit_A$ as in Section 3.4. We will design **RsvScheduler** and **CmtScheduler** to generate rsv and cmt signals which will reserve and commit as many rules as possible while maintaining correctness.

Note: we are using a short hand of rsv for “reserve”, cmt for “commit” and rc for “ready to commit”.

3.6 Greedy Rsv Scheduler and Cmt Scheduler for the GROC Scheduler

Consider a scheduler that initially issues rsv commands for a non-conflicting set of rules and simultaneously resets the **CyclesLeft** counter for these rules. Whenever a rule commits, the scheduler can issue additional reservations for some rules such that the new rules do not conflict with the rules still holding reservations. This scheduler can be improved by resetting the **CyclesLeft** counter only when some commit affects the input of the rule and not at the time of reservation. As we will show, the GROC scheduler will allow opportunistic commits, that is, rc_A may be asserted even before the rule ever becomes active. However, the **Rsv Scheduler** of the GROC scheduler will not issue a cmt_A command unless rule A holds a reservation. The computation of the rsv and cmt signals is a bit complicated but we will ensure that each execution of a rule A corresponds to the state of the input registers during a precisely defined clock cycle: the cycle in which the rsv_A signal is asserted. Earlier we have called this clock cycle the rule activation cycle. The results of an active rule are committed to the global state in the clock cycle when cmt_A signal is asserted. The subtlety is that the time duration between cmt_A and rsv_A may be less than L_A .

3.7 Scheduling Invariants

The GROC scheduler preserves the three invariants in Figure 3-6 in order to guarantee atomic execution of the rules. We will now provide the motivation behind these invariants.

Invariant 1. if $A <> B$, a reservation for A cannot be issued if B already holds a reservation;
Invariant 2. if $A < B$, B cannot commit while A holds a reservation, i.e., is active;
Invariant 3. if $A < B$, a reservation for A cannot be issued if B holds a reservation, i.e., is already active;

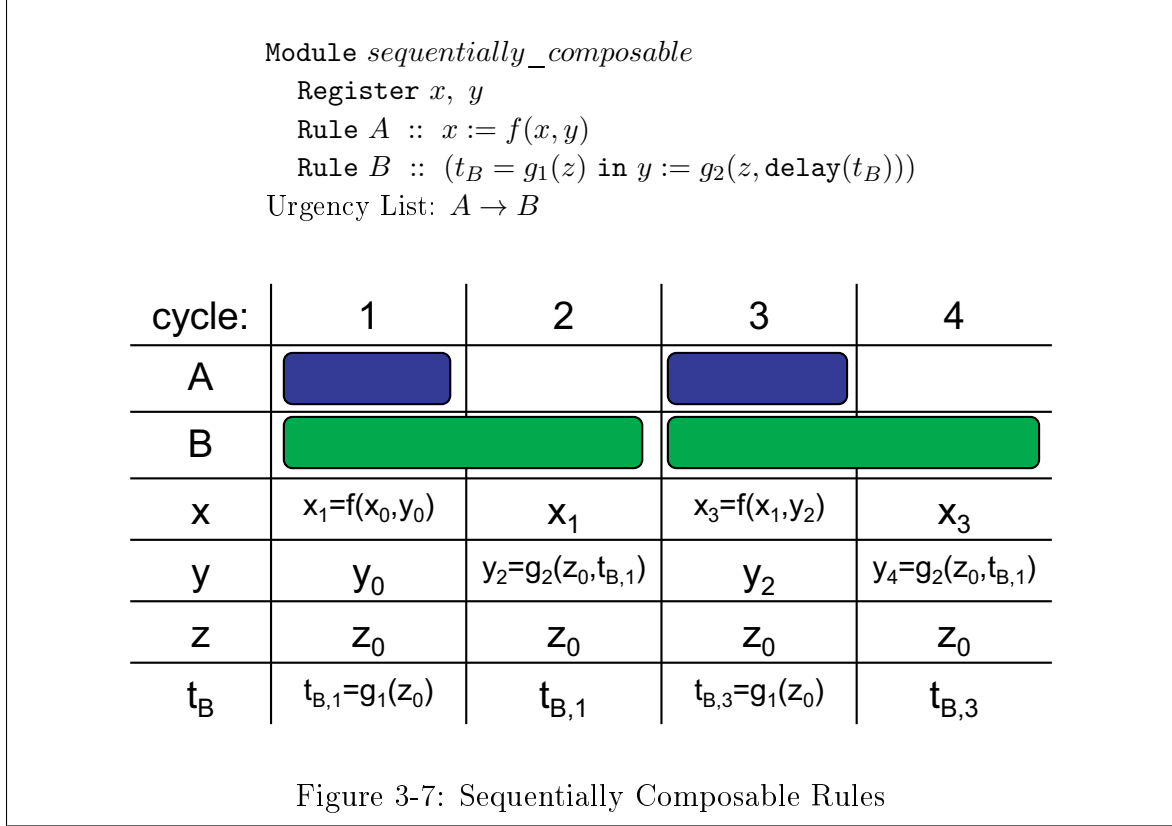
Figure 3-6: GROC Scheduler Invariants

Execution of Conflicting Rules: Invariant 1 is obvious – concurrent scheduling of conflicting rules is guaranteed to lead to atomicity violation. Conflicting rules must be executed one after another.

Rule Stretching: Sequentially composable rules are characterized by having one rule read the output of the other, but not vice versa. Consider the example of two sequentially composable rules shown in Figure 3-7. Figure 3-7 demonstrates that it is possible for two SC rules to be active concurrently and still produce results consistent with atomic execution. In our example, the schedule of firing is A, B, A, B . Notice that even though A and B start in the same clock cycle, this is the only schedule which produces the final state in Figure 3-7.

When executing SC rules it is sometimes necessary to keep a rule active longer (delay its commit) than the expected number of clock cycles. We call this *rule stretching*. An example of rule stretching is demonstrated in Figure 3-8. The Figure demonstrates that committing rule B too early can lead to atomically inconsistent results of computation. This is the motivation underlying Invariant 2.

Livelock Avoidance: Consider the example shown in Figure 3-9. If all three rules are activated in the same clock cycle it will lead to a livelock, because Invariant 2 would not allow any of the three rules to commit. The issue here is that rules A , B and C form a dependency cycle. The Hoe-Arvind scheduler [29] avoided this problem by disallowing the concurrent scheduling of such cyclically dependent rules. Their



scheduler permitted scheduling of only a subset of dependency-cycle-forming rules. Though such subsets are not unique, finding one is easy, especially given the Urgency List. One can simply delete the least urgent rule that breaks the dependency cycle and schedule the rest.

In the multi-cycle execution model, the problem is slightly harder because such cyclically dependent rules may not all begin at the same time. For example, as shown in 3-9b) the rules B and C may already be executing when one has to decide whether to activate rule A . If A is activated it will lead to a livelock. This is the motivation underlying Invariant 3 which prohibits such an activation of rule A . It is worth noting that Invariants 1 and 3 guarantee that active rules at any time form a partial order with respect to the sequential composability ($<$) relation. Consequently, it is always possible to pick an active rule which can be committed before all others without violating atomicity.

Our Invariant 3 is not the most permissive invariant because it may prevent some legal schedules. For example, in Figure 3-7 it is possible to fire rule A in clock cycles 2

Module *rule_stretching*

Register x, y, z

Rule $A :: (t_A := f_1(y) \text{ in } x := f_2(y, \text{delay}(t_A)))$

Rule $B :: y := g(z)$

Urgency List: $A \rightarrow B$

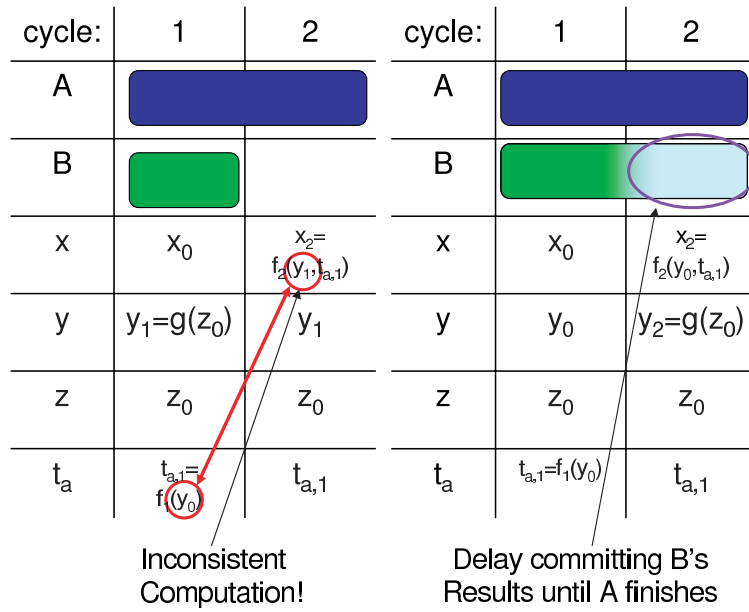


Figure 3-8: Rule Stretching (Delayed Commit)

Module *deadlock*







Register x, y, z

Rule $A :: x := f(y)$






Rule $B :: y := g(z)$

Rule $C :: z := h(x)$

Urgency List: $A \rightarrow B \rightarrow C$

cycle:	1	2	3
A			
B			
C			
x	x_0	x_0	x_0
y	y_0	y_0	y_0
z	z_0	z_0	z_0

a)

cycle:	1	2	3
A			
B			
C			
x	x_0	x_0	x_0
y	y_0	y_0	y_0
z	z_0	z_0	z_0

b)

Since $A < B$, $B < C$ and $C < A$, Invariant 2 stops any rule from committing.

Figure 3-9: Livelock avoidance

and 4, resulting in the execution A, A, B, A, A, B . We will not explore such aggressive schedules in this thesis.

3.8 Scheduling Algorithm

We now present the procedures to generate the rsv_A and cmt_A signals based on the invariants we have discussed in Section 3.7. Both procedures rely on the compiler-computed conflict properties for each pair of rules. The **RsvScheduler** procedure generates the rsv signals and requires the sA and $mayRsv$ input signals while the **CmtScheduler** procedure generates the cmt signals and requires the rsv generated by **RsvScheduler** and rc signals. In the current version of the scheduler, $mayRsv$ signal is equivalent to rsv , as shown in Figure 3-5, but that will change in Chapter 5.

In order to set rsv_A at time t we need to make sure that $mayRsv_A(t)$ is *true* and that A does not conflict with the rules that have already been reserved. Furthermore, if sA_A is asserted, we must assert rsv_A . The rules are examined in the decreasing order of urgency. The algorithm in Figure 3-10 builds the set of active rules AR and a set of non-active rules NA . Step 3.2 of the algorithm above preserves Invariants 1 and 3. As a consequence the rules in AR will form a partial order over the sequentially composable ($<$) relation.

Procedure RsvScheduler:

1. Initially $AR = \{A \mid sA_A(t)\}$ and $NA = \mathcal{R} - AR$
2. $\forall A \in AR : rsv_A(t) = true$
3. repeat while $NA \neq \emptyset$
 - 3.1 Let A be the most urgent rule in NA : $\nexists B \in NA : B \rightarrow A$
 - 3.2 $rsv_A(t) = mayRsv_A(t) \wedge (\nexists B \in AR : ((A <> B) \vee (A < B)))$
 - 3.3 $NA = NA - \{A\}$
 - 3.4 if $rsv_A(t)$ then $AR = AR \cup \{A\}$

Figure 3-10: **RsvScheduler** Procedure

In order to compute $cmt_A(t)$ we need to make sure that 1. $rsv_A(t)$ is *true*; 2. $rc_A(t)$ is *true* and 3. committing A would not violate Invariant 2. The algorithm in Figure 3-11 uses the set AR computed in Figure 3-10 and the set CR which is the set of rules which are active and ready to commit. It examines the rules in CR in

Procedure CmtScheduler:

1. Initially $CR = \{A \mid active_A(t) \wedge rc_A(t)\}$; $AR = \{A \mid rsv_A(t)\}$
2. $\forall A \in \mathcal{R} - CR : cmt_A(t) = false$
3. Repeat while $CR \neq \emptyset$
 - 3.1 Select $A \in CR$ such that $\nexists B \in CR : B < A$
 - 3.2 $cmt_A(t) = \nexists B \in AR : (B < A)$
 - 3.3 $CR = CR - \{A\}$
 - 3.4 if $cmt_A(t)$ then $AR = AR - \{A\}$

Figure 3-11: **CmtScheduler** Procedure

an order defined by sequential composability ($<$) and decides for each rule whether it should be allowed to commit:

It is important to note that this algorithm is well defined (deadlock free) only if the ($<$) relation is a partial order on CR. This is not true in general but is true for active rules because of Invariants 1 and 2 and because CR is a subset of AR .

These algorithms can be turned into pure combinational logic for a given set of rules because ($<>$) and ($<$) relation among rules are known at compile time. We will illustrate this in Section 3.10 using the GCD example.

We will like to note one point about the efficiency of **CmtScheduler** procedure. Even though the **CmtScheduler** procedure is efficient – it generates its output in a single pass over the set CR – it can potentially generate a lot of combinational logic because of step 3.1. This step essentially requires a topological sort of any subset of rules based on the ($<$) relation. This inefficiency can be overcome by a version of the algorithm that sometimes may not commit as many rules every clock cycle as theoretically possible.

The simplification of the algorithm is based on the fact that the rules in step 3.1 can be examined in any order, that is, we can ignore the ($B < A$) check without the loss of correctness. So instead of checking rules according to the ($<$) relation, we can check the rules in the urgency order, which is very efficient to implement in hardware.

3.9 Correctness of GROC Scheduler

It is fairly easy to show that the behavior of circuits scheduled using the GROC scheduler can be duplicated by a one rule at-a-time scheduler as described in Section 3.2. Since one rule at-a-time scheduler is the model for guarded atomic action execution, this would verify correctness of GROC.

We observe that step 3.2 of **RsvScheduler** procedure guarantees that we only activate rules which do not read the result of any already active rule. Next we observe that step 3.2 of **CmtScheduler** procedure commits only active rules, and commits them in an order such that there are no other active rules which read the input of the rule being committed. Together these two observations mean that the domain of an active rule cannot change after it has been reserved and before it has been committed.

Thus if we start both schedulers with the same state, and look at a single clock cycle of execution of the GROC scheduler, we can schedule rules one at-a-time in the order in which they are committed in step 3.2 of **CmtScheduler** procedure and we must arrive to the exact same state as we would if we committed all these rules in a single clock cycle using the GROC scheduler.

Thus, by induction, it must be the case that if we start with the same initial state, we will be able to duplicate the behavior of the GROC schedule with a one rule at-a-time scheduler.

3.10 The GROC Scheduler for GCD

The GCD example in Section 3 uses Bluespec methods, which our scheduler does not accommodate. In order to use this example here, we modified the module to produce test data and print results. We present the complete example in Figure 3-12. The complete module consists of three rules: *GCD*, *Seed*, and *Print* (abbreviated to *G*, *S* and *P*). Those three rules have the following interactions: $G \lt \! > S$, $P \lt G$ and $P \lt S$. Of course, the two rules which update their data are also self-conflicting: $G \lt \! > G$ and $S \lt \! > S$.


```

Module doGcd2
  Register  $x = 1, y = 0, a = 1, b = 1, t$ 
  Rule GCD ::  $(x := y \mid (t := \%_1(x, y) ; y := \%_2(x, y, t)))$  when  $y \neq 0$ 
  Rule Seed ::  $(x := a \mid y := b \mid a := a + b \mid b := a)$  when  $y = 0$ 
  Rule Print ::  $\$display(a + "\%" + b + " = " + x)$  when  $y = 0$ 

Urgency List: Print  $\rightarrow$  GCD  $\rightarrow$  Seed

```

Figure 3-12: Complete GCD Example

$$\begin{aligned}
rsv_G(t) &:= sA_G(t) \wedge (rdy_G(t) \wedge \neg sA_S(t)) \\
rsv_S(t) &:= sA_S(t) \wedge (rdy_S(t) \wedge \neg rsv_G(t)) \\
rsv_P(t) &:= sA_P(t) \wedge (rdy_P(t) \wedge \neg sA_G(t) \wedge \neg sA_S(t)) \\
cmt_G(t) &:= rsv_G(t) \wedge rc_G(t) \wedge \neg(rsv_P(t) \wedge \neg cmt_P(t)) \\
cmt_S(t) &:= rsv_S(t) \wedge rc_S(t) \wedge \neg(rsv_P(t) \wedge \neg cmt_P(t)) \\
cmt_P(t) &:= rsv_P(t) \wedge rc_P(t)
\end{aligned}$$

Figure 3-13: Complete GCD Control and Scheduling Equations

During creation of the *rsv* and *cmt* signals for the module, we have to take care to respect the Invariants, as described in Section 3.8. *rsv* signals for *G* and *S* have to respect Invariant 1 ($G <> S$), while *P* has to respect Invariant 3 ($P < S$ and $P < G$). *cmt* signals for *G* and *S* rules have to respect Invariant 2 ($P < S$ and $P < G$).

The complete set of control and scheduling equations for our GCD example is in Figure 3-13.

Chapter 4

Flat, Flexible, Multi-Cycle Scheduling

In this chapter we will relax the requirement that every rule complete its computation in a constant, statically-known number of clock cycles. This will allow us to develop circuits for operators such as a dynamic conditional (non-strict `if`) and a while-loop. Guards of our new circuits will be permitted to be multi-cycle, something which we did not permit before. We will begin with motivating our work by modifying our old GCD example to use a loop to perform its computation. We will then introduce a three-wire data format, which is used for dynamically tracking propagation of data, followed by a presentation of circuits for compilation of all actions and operators available in flat BMC using syntax-directed translation. Finally, we will show how this new compilation scheme fits with the GROC scheduler from the previous chapter. This chapter still only deals with flat modules.

4.1 Loop-based GCD Example

Consider the GCD design presented in Figure 4-1. We have eliminated the split-phase request-response design from Figures 2-2 and 3-1 and replaced it with an imperative function. The user can simply perform a single value method call to the GCD method, passing in the parameters to the computation, and the answer will be provided in the same atomic action. This style of computation can lead to simpler high-level design, as the synchronization of the computation is implicit rather than explicit. The power

of the request-response design is not always necessary.

Note that the number of clock cycles the computation takes is unbounded. While it is possible to compute the absolute maximum number of clock cycles for particular sized inputs, this would be almost certainly unduly pessimistic for many input vectors. Our goal is to create a methodology to synthesize circuits which will adjust their computation time according to dynamic data, rather than statically computed worst-case scenarios, as in the previous chapter.

Figure 4-2 shows a circuit that implements the loop-based version of the GCD. The computation is started by asserting en_{GCD} for one clock cycle at which point the inputs are captured. Completion of the computation is signaled by asserting $done_{GCD}$ signal and the result is presented on $result_{GCD}$ wire. The inputs are first captured in the X and Y flip-flops, and the next clock cycle the computation begins. The computation is finished when value stored in the Y flip-flop is 0.

Aside from the $done_{GCD}$ wire, the interface is similar to that used by “regular” Bluespec. This additional wire is similar to the $rdyCommit$ wire from previous chapter, but signals the completion of computation of a value method rather than an action.

```
Module doGcd3
  ValMeth GCD(x,y)  ::  while (y ≠ 0) do
                        (x,y) = ( if (x ≥ y) then (x - y, y) else (y - x, x) )
                        return x
```

Figure 4-1: Loop-based GCD example

4.2 (*Valid, Changed, Data*) 3-tuples

In the previous chapter we have introduced the `CyclesLeft` counter which we used to keep track of propagation of data through bodies of rules. Our counters had a static reset value and would count down to 0, at which point we were guaranteed that the rule has processed all its data and is ready to commit.

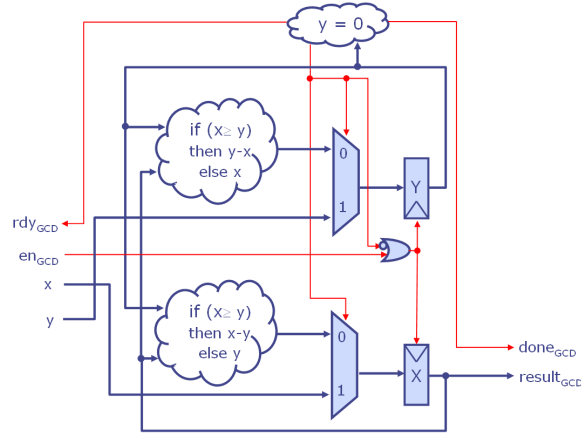


Figure 4-2: Loop-based GCD Circuit

In this chapter we want to extend our approach to allow us to generate rules with dynamically determined completion times. That is we would like to create rules which can incorporate loops, such as the GCD in Figure 4-1 and conditionals which take advantage of unbalanced paths. A simple counter cannot provide the required flexibility.

One approach to solving this problem is to generate a *valid* signal that travels along with the data. The *valid* signals indicate that the data observed on the wire matches the data currently stored in the architectural state. For every clock cycle delay on the data path, the *valid* signal is also delayed. When two data signals are being combined by an operator, the output's *valid* signal cannot be asserted until both inputs have their *valid* signals asserted.

This is a good approach, but is not sufficient for our purposes. In order to truly tell us whether the data on the wire represents the architectural state, the *valid* signals must be immediately reset when the architectural state changes. To accomplish that, we also generate a *changed* signal. We will assert the *changed* signal for one clock cycle immediately after the architectural state has been changed. We will use this signal to reset our *valid* signals.

In other words, there are really three states that the data on a wire can be in: not valid, newly valid with new data (first clock cycle with this data), valid with old data

(not changed from last cycle). We need to distinguish between the two different valid states because the consumer of the data needs to know if it should be recalculating its result based on new values. To communicate these three states, we also generate a *changed* signal.

Our notation will be to associate a 3-tuple (x_v, x_{ch}, x_d) with every variable x that represents an expression. x_d represents the actual value computed by the expression. The *valid* signal x_v indicates that the expression has finished computing, and that the results are consistent with the current architectural state of the system. The *changed* signal x_{ch} indicates that the architectural state has just changed in the previous clock cycle and the expression has to be recomputed.

One important invariant that comes from the description of the 3-tuple is that once the data becomes valid, meaning x_v is asserted, both the data wire (x_d) and the *valid* signal (x_v) must remain constant until the *changed* signal (x_{ch}) is asserted.

In our figures, we draw a single blue wire for these 3-tuples. We continue the scheme of drawing boolean values with thin lines and other values with thick lines.

4.3 Architectural Registers

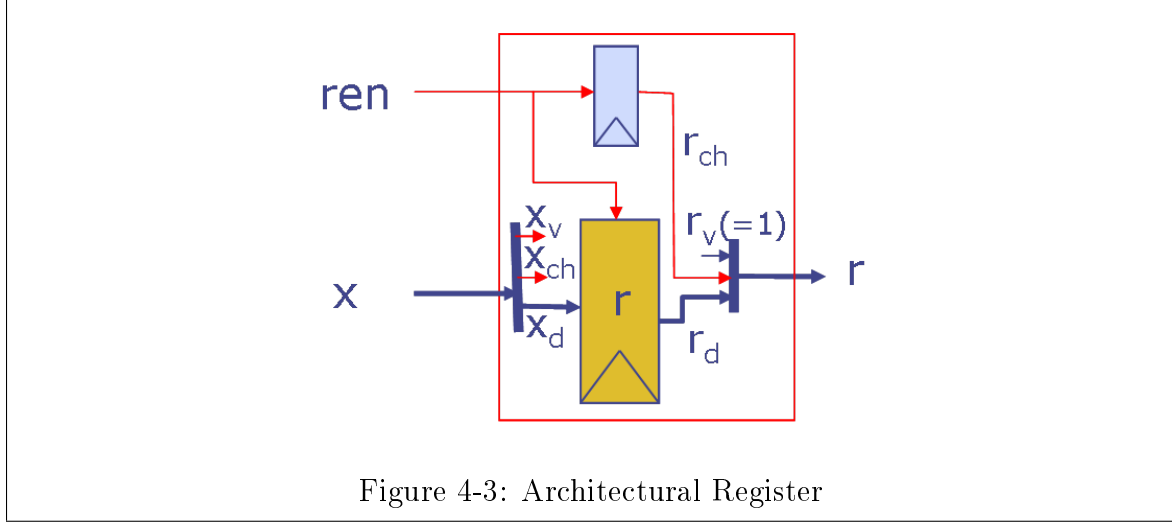
Values of architectural registers are used in expressions and are written with the result of evaluation of an expression. Since expressions consume and produce 3-tuples described in the last section, we must create circuits that will do the same. In addition, architectural registers are only updated on selected clock cycles, so we must also take care to incorporate the register write enable signal (*ren*).

On the input side, we can simply drop non-data part of the 3-tuple. We will build circuits which will guarantee that we write only valid data to architectural registers.

A value being read from an architectural register is always valid. That means for any register r , r_v is necessarily 1 at all times.

By definition, the data read from an architectural register is updated in the clock cycle after it has been written. Thus $r(t)_{ch} = ren(t - 1)$.

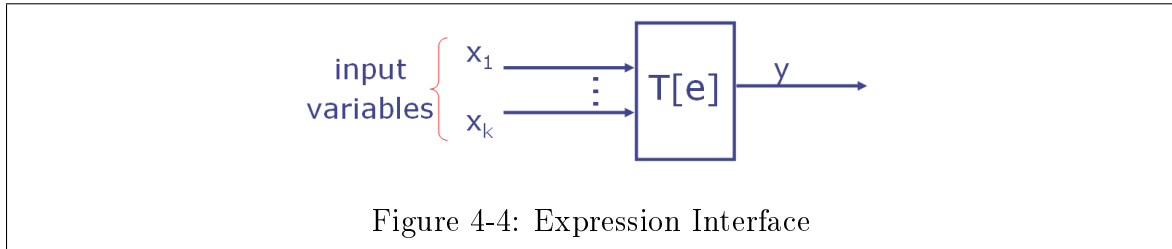
Figure 4-3 shows the hardware circuit generated by an architectural register.



4.4 Syntax-directed Translation of Multi-cycle Expressions

In this section we will present our syntax-directed scheme for generating circuits for multi-cycle expressions.

4.4.1 Expression Interface



Every expression has the interface as shown in Figure 4-4. The inputs and outputs are all 3-tuples (x_d, x_v, x_{ch}) . In the this and following figures, $T[e]$ represents the hardware circuit generated for expression e .

4.4.2 Register, Constant and Variable

Figure 4-5 shows the hardware circuit generated by references to registers, constants or variables. The circuit simply returns the 3-tuple represented the constant c or the

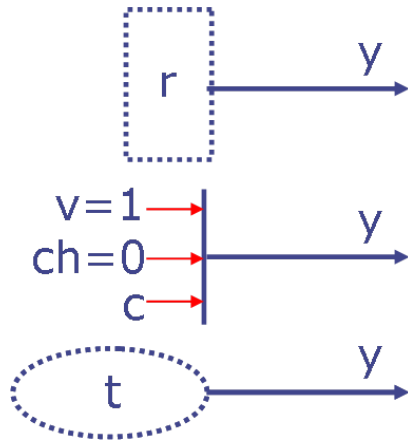


Figure 4-5: Register, Constant and Variable

variable t . Note that for a constant c , the *valid* signal always remains *true* and the *changed* signal always remains *false*.

The register reference gets all of its signals directly from the circuit for the architectural register, which we will present in Section 4.3.

4.4.3 Let Expression

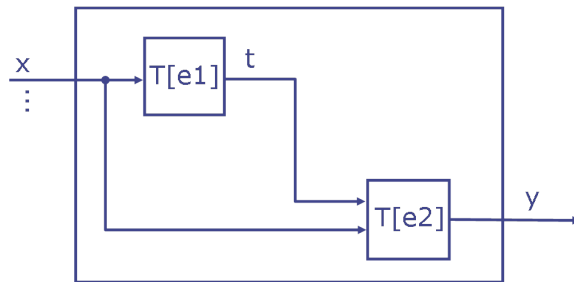


Figure 4-6: Let Expression

Figure 4-6 shows the hardware circuit generated by a let expression. The expression $e1$ is bound to variable t and is sent as input to expression $e2$.

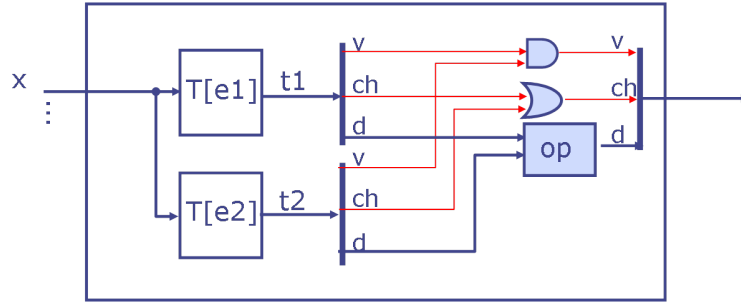


Figure 4-7: Primitive Operator

4.4.4 Primitive Operator

Figure 4-7 shows the hardware circuit generated by a primitive operator. The output can be computed only when both the expressions have been computed (conjunction of the *valid* signals in Figure 4-7). Any expression depending on this primitive operator must be recomputed whenever either of the expressions in the primitive operator are recomputed (disjunction of the *changed* signals in Figure 4-7).

4.4.5 Conditional Expression

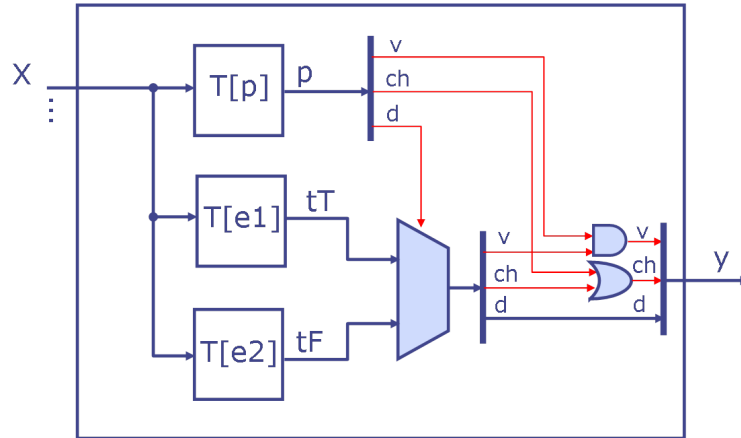


Figure 4-8: Conditional Expression

Figure 4-8 shows the hardware circuit generated by a conditional expression. The circuit waits for the predicate p to become valid. Then, depending on the value of the

predicate, it waits for either the true or the false expression to become valid. The circuit thus waits only for one of the two branches of the conditional to become valid, depending on the predicate. Thus this condition expression is a non-strict **if**. Any expression that depends on this conditional expression must be recomputed whenever p is recomputed or, depending on the value of the predicate, either the true or the false expression is recomputed.

4.4.6 Delay Expression

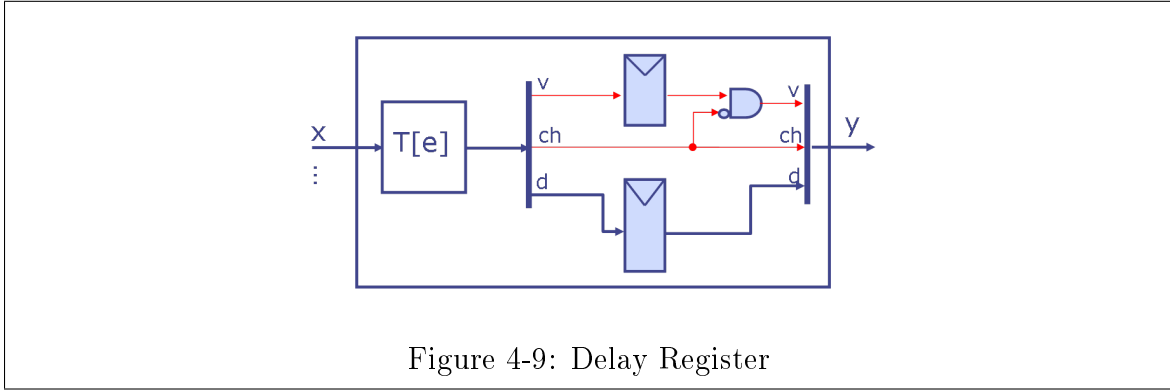


Figure 4-9: Delay Register

Figure 4-9 shows the hardware circuit generated by a delay expression. The value of expression e along with its *valid* signal gets registered. Whenever e is recomputed, the *changed* signal for e is set and hence the value produced by **delay**(e) becomes invalid. The *changed* signal is passed on without being registered because it has to invalidate the values produced by all the expressions that use the value produced by **delay**(e). Once e is recomputed, **delay**(e) produces a valid value, which gets passed on to further expressions.

As explained earlier, the use of a flip-flop to register the data portion of the expression is optional. We could just as well declare the path to be a multi-cycle combinational path and skip the flip-flop.

It is possible that converting a complex expression to execute over many clock cycles using the delay operator would create a long combinational path on the changed signal. We discuss this possibility in Section 7.5.2.

4.4.7 While Expression

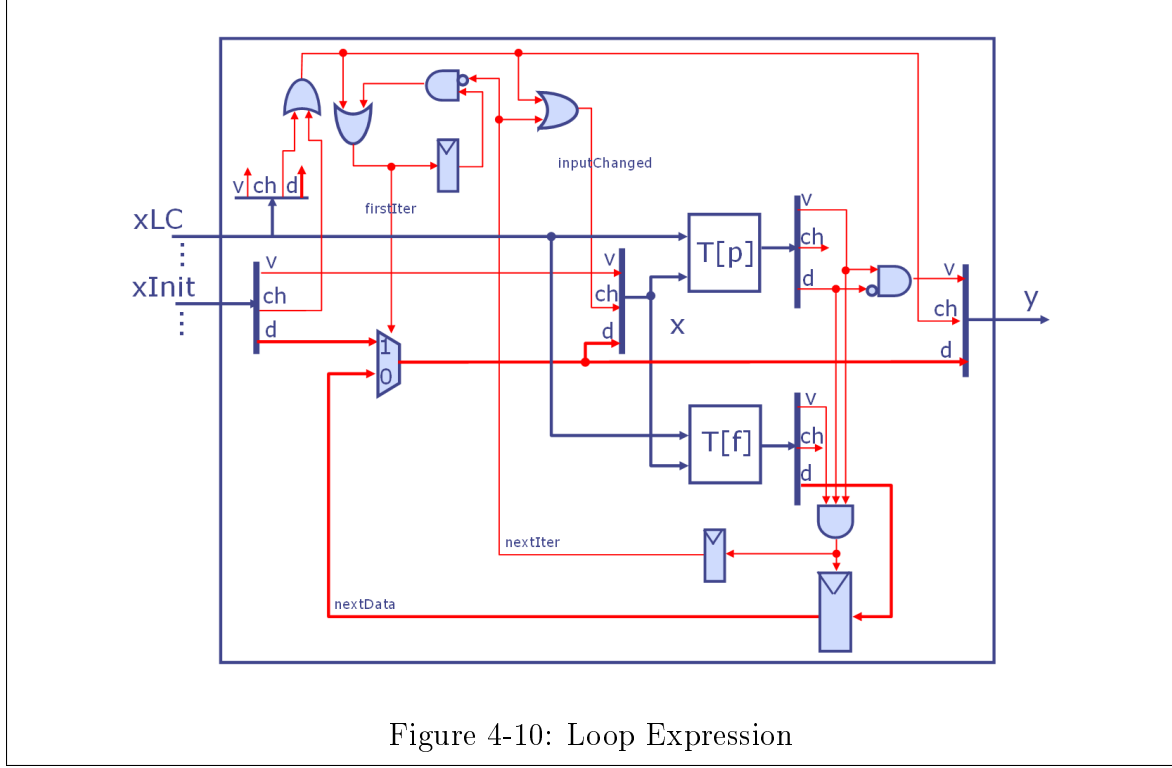


Figure 4-10: Loop Expression

Figure 4-10 shows the hardware circuit generated by a while expression. The while expression introduces a micro-architectural register to store the values produced in each successive iteration of the loop as shown in the figure. $xInit$ is the loop-dependent input to the loop, while xLC is the loop constant input. x is the internal loop variable.

In the first iteration of the loop, the value of $xInit$ is taken from the input to the while expression; during the subsequent iterations, the value is taken from the loop iteration register ($nextData$). The $firstIter$ signal denotes if the loop is executing the first iteration or not. The loop repeats its computation as long as the predicate p for the loop evaluates to *false*. Since the predicate itself can take x as input, the number of iterations of the loop will not be known at compile time. The signal $nextIter$ behaves like a pulse in the sense that it is asserted for one clock cycle only after every iteration, thus denoting whether the loop is starting the next iteration. $nextIter$, xLC_{ch} and $xInit_{ch}$ are combined to generate the $inputChanged$ signal

which indicates the restart the computation in the predicate p and function f for every iteration of the loop. This is done by asserting x_{ch} signal. The loop iterations continue till the predicate becomes *false*, after which the loop produces the valid result.

If at any time either of the inputs to the loop changes by asserting their *changed* signals, we immediately reset the execution of the loop. As long as the inputs' *changed* signals remain low, the loop will continue its computation until it comes to a completion.

Once the predicate evaluates to *true*, we stop iterating by preventing the micro-architectural register from capturing next value of the loop variable.

4.5 Syntax-directed Translation of Multi-cycle Actions

In this section we will describe our scheme for generating circuits for multi-cycle actions, given that all expressions have already been generated using the technique described in previous section.

4.5.1 Action Interface

The **Rule Selector** circuit described in Chapter 3 uses *rdyCommit* signals to indicate that the body of a rule (the rule's action) is finished performing its computation and is ready to commit. We would like to use the same interface as the Rule Selector, so we will need to also generate the *rdyCommit* signals from our actions. In this chapter we will rename *rdyCommit* signals to just *rc* for simplicity.

While our language has a variety of different actions available for creating designs, there is only one action that directly changes the architectural state of the machine: register updates. Thus every action needs to generate signals containing the new values being written to the registers. We will call these signals *rd*. *rd* signals do not need to be 3-tuple, because no further computation is being done on them. The data

they carry will simply be written to a register.

Our language also contains the conditional action, which allows us to perform the underlying action conditionally. This means that the same action can skip a register write, depending on the result of an execution-check. Thus we will marry the register update data signals with a register write predicate, rp , which will tell us if the write needs to actually take place. rp wires also do not need to be 3-tuple, because they will only participate in muxing of the correct register data.

Together rp and rd form a tuple (rp, rd) .

The interface for multi-cycle actions is presented in Figure 4-11. In the this and following figures, $T[a]$ represents the hardware circuit generated for action a .

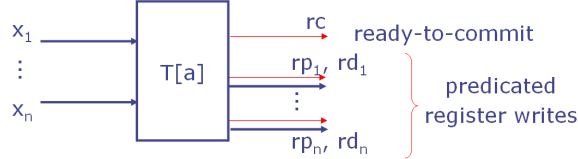


Figure 4-11: Action Interface

4.5.2 Register Assignment

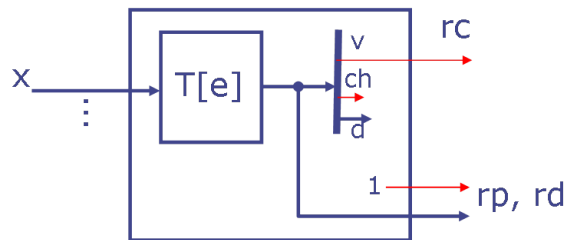


Figure 4-12: Register Write Action

Figure 4-12 shows the hardware circuit generated by a primitive register write action. The ready-to-commit signal (rc) is generated whenever the expression which produces the value for the register becomes valid. Since this action is not predicated, rp is set to *true*.

4.5.3 Conditional Action

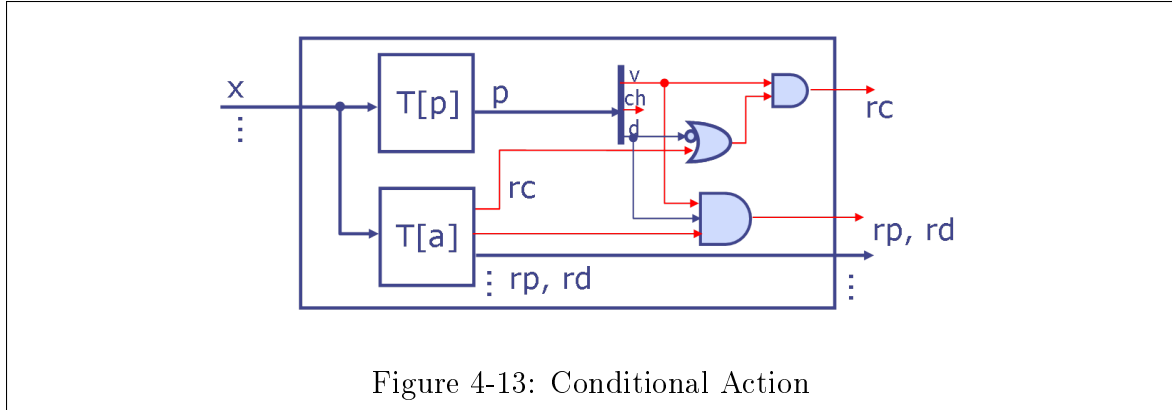


Figure 4-13 shows a conditional action. This action must wait for the predicate p to become valid. Then, if the predicate is *false*, the action is ready to commit irrespective of whether the action a is ready to commit. But if the predicate is *true*, then this action must wait for a to become ready to commit. The ready-to-commit (rc) signal shown in the figure reflects this. All the register updates of action a must happen only if the predicate p of this action has been computed and it is *true*.

4.5.4 Parallel Action

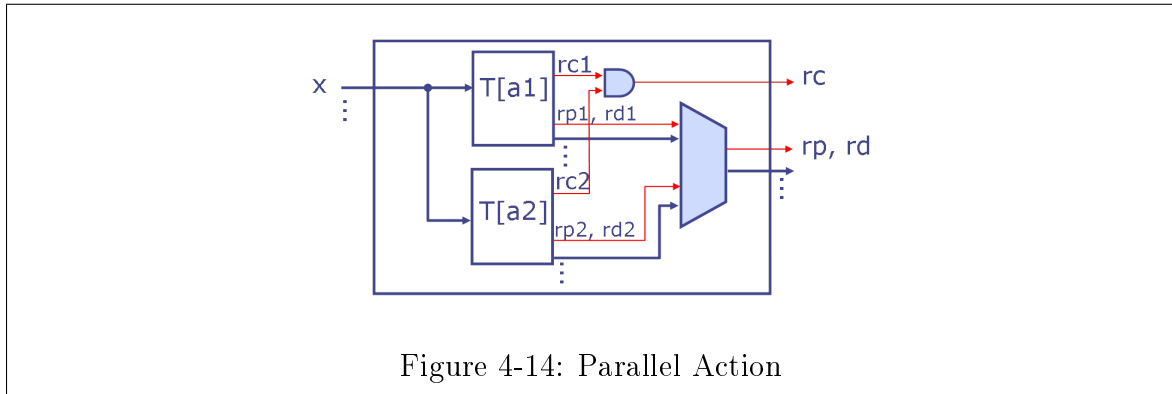


Figure 4-14 shows a parallel action which is the parallel composition of two actions. The whole action is ready to commit only when both the actions are ready to commit. The MUX shown in the figure composes the register updates of the two actions. If both the actions are writing to the same register, then it is an error if the predicates

for that register in both the actions are *true*. For each register, the MUX passes on the predicate and the update-expression from the action where the predicate to update that register is *true*. The MUX is a simple one-hot-MUX with *rp* as the selectors.

4.5.5 Let Action

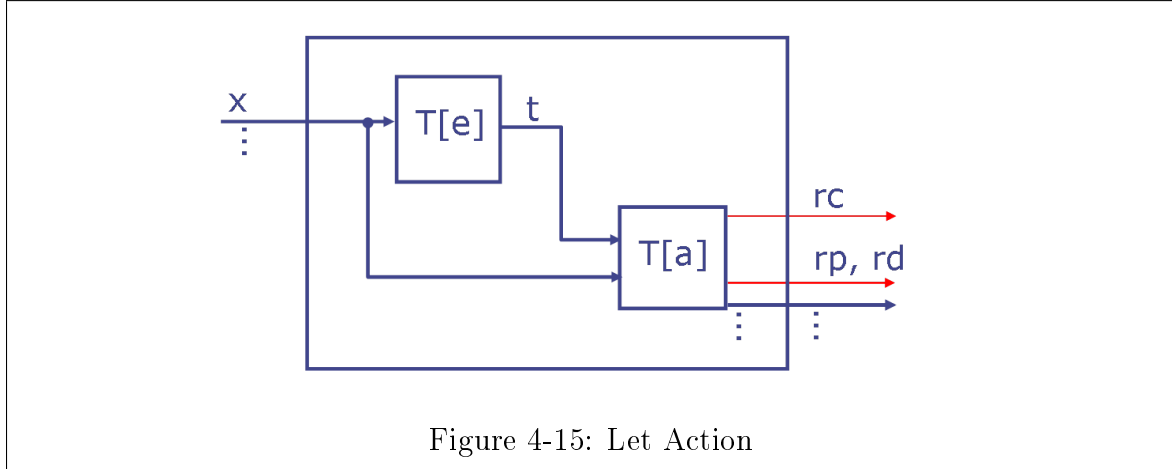


Figure 4-15 shows the hardware circuit generated by a let action. The expression *e* is bound to variable *t* and is sent as input to action *a*.

4.6 Syntax-directed Translation of Rules

4.6.1 Rule Interface

A rule in our language consists of an expression guard and an action. Thus the interface for a rule must read architectural state (for both the guard and the action), produce a *rdy* signal from the guard, *rc* signal from the action, and a set of (*rp*, *rd*) tuples also from the rule. Figure 4-16 presents the interface of a rule.

4.6.2 Rule Circuit

Most of the signals for a rule are generated directly from the guard and the action. The only exception is the *rdy* signal. The *rdy* signal is produced from the result of

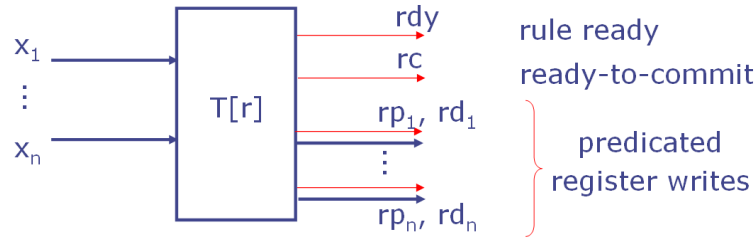


Figure 4-16: Rule Interface

evaluating the guard but is is a single wire, not a 3-tuple. *rdy* should be asserted when the guard evaluates to *true*. Thus it must be equal to the conjunction of the data and valid signals from the guard evaluation circuit.

The rule circuit is presented in Figure 4-17.

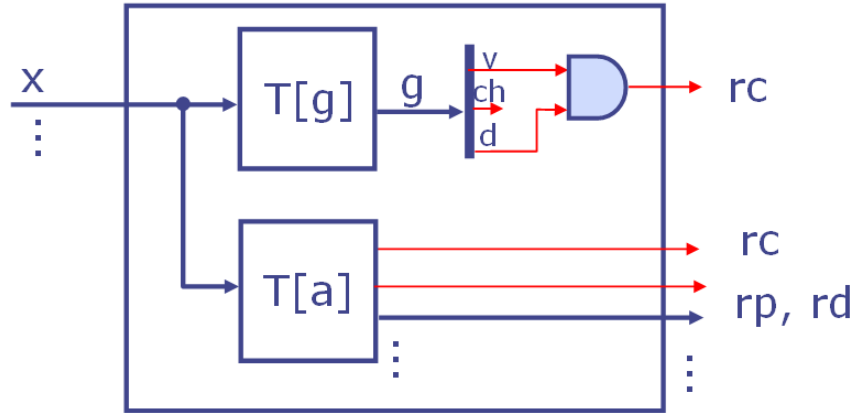
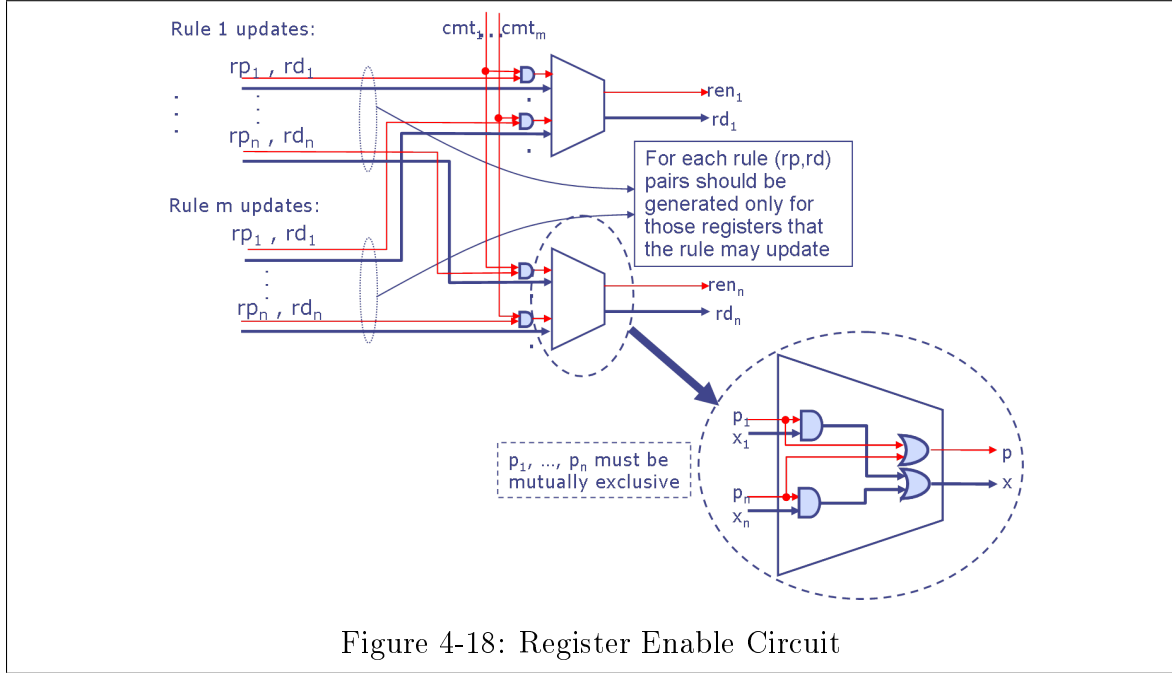


Figure 4-17: Rule Circuit

4.6.3 Combining Rules for Register Writes

In our compilation scheme, each rule is compiled separately. All the signals consumed are always available to all rules. The *rdy* and *rc* signals are passed to the scheduler without modification, and never shared between rules. But the register write signals (*rp*, *rd*) need to be explicitly combined in the case that two rules may update the same state.

Our scheduler will guarantee that for any architectural register, only one rule will try to update it in a single clock cycle. Figure 4-18 presents our circuit for generating the final *ren* and *rd* signals which will be fed directly to the architectural register circuit from Section 4.3. First, we filter out the *rp* signals to only take into account those which belong to committing rules. We then pass the register data through a one-hot MUX.

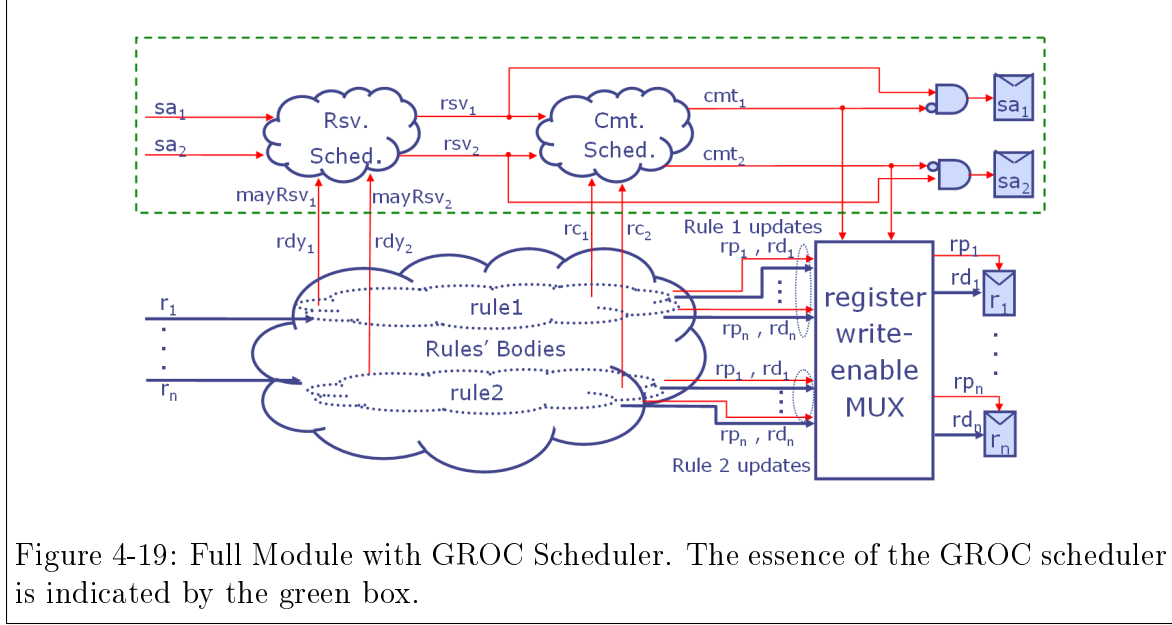


4.7 Module Model

We have now introduced all the major components of our multi-cycle modules and can present the top-level model in Figure 4-19. It is important to note that the essence of the GROC scheduler from Figure 4-19 has not been modified. This means that we are using the same **RsvScheduler** and **CmtScheduler** algorithms from Section 3.8.

4.7.1 Single-cycle Equivalency

In the case where the design contains no delay registers or loops in the design, our circuits can be optimized easily to generate the same hardware as the single-cycle



Bluespec designs. In the absence of delays, all values communicated through our 3-tuples are always valid. This is because even when an architectural register is updated, the updated value is visible throughout the circuit the very next clock cycle. That means that all the logic for computing *valid* signals can be optimized out. Similarly, because the only use of *changed* signals is to compute multi-cycle *valid* signal, those can also be optimized out, together with the one-bit registers that accompany architectural registers. Finally, actions are always ready to commit, because all expressions are always providing valid outputs. Thus all the *sa* bits from the scheduler will always hold 0, and are also subject to elimination. Without *sa* bits, the scheduler becomes the standard Bluespec scheduler [29].

Chapter 5

Flat Scheduling In Presence of Guards

In this chapter we will enhance the syntax directed synthesis scheme from Chapter 4 to include guards for all expressions and actions. As before, the guards can be multi-cycle. In order to accommodate slow guards while still giving the user effective control over the urgency list, we will turn make the scheduler speculative.

5.1 Speculative Scheduler

In the previous chapters we have presented a GROC scheduler based on two-phase scheduling: scheduling reservations and commits separately (**RsvScheduler** and **CmtScheduler** algorithms). As described, the approach relies on the ability to evaluate the *rdy* signal accurately before reserving a rule in the first step. However, with the introduction of multi-cycle guards, we can no longer guarantee that all guards will ever complete their evaluation. In this chapter we are also transforming the construction of actions and expressions to allow for explicit guards, rather than force the lifting of the guards to the overall guard for the encompassing rule. This means that it will be easier for the designer to construct designs with complex guard expressions, which may require multi-cycle implementations to perform well.

Consider a module with two rules: one has a slow multi-cycle guard, and the

other rule is has a single-cycle guard, is always ready and updates a register used in computation of the guard for the first rule. In such a scenario, the second rule will continually be reserved, commit and be reserved the very next clock cycle. The guard for the first rule will never be given a chance to fully evaluate, so the rule will never receive a reservation. This situation can be a serious problem if the first rule should receive higher urgency than the second rule.

In order to overcome this problem, we have designed a variation on the GROC scheduler. In the previous chapter we assumed an unevaluated guard to be *false*. Starting with this chapter we will consider an unevaluated guard to be *false*. This allows us to reserve rules speculatively, that is before we have fully evaluated the guard. During the evaluation of the rule, we will fully evaluate the guard before committing. If the guard evaluates to *false*, we release the reservation, and take care not to reserve the rule again until its guard has been updated by another rule.

Since guards can now evaluate over multiple clock cycles, we must also consider how we can tell that a guard has been fully evaluated. After all, with multi-cycle guards, a boolean guard can now really have three states: *true*, *false* and *unknown*. In the previous chapter, we used the *rdy* signal, which was asserted when the guard was evaluated to true but remained unasserted for both *false* and unknown states. The speculative reservation scheme described above requires that we be able to tell when the guard has evaluated to *false* (so we can release existing reservation and block future reservations until the guard changes) and when it hasn't (thus is *true* or *unknown*). In order to facilitate this distinction we will propagate the value of the guard using a *notRdy* signal (as opposed to a *rdy* signal before). *notRdy* will be asserted when the guard evaluates to *false*, and remain unasserted otherwise. If *notRdy* is asserted, we must release the reservation on the corresponding rule (if we have issued one) and avoid reserving the rule until the guard has been changed.

While we can simply use *notRdy* to determine when we must release the reservation for a rule due to its guard evaluating to *false*, we also need some way to determine when the guard has actually evaluated to *true*. This is because we cannot commit a rule that has not fully evaluated its guard to *true*. This is where we will

utilize the *rc* signal we have used in previous chapters. Our circuits will guarantee that if *rc* is asserted then the guard is fully evaluated, and if *notRdy* is not asserted then the guard is *true*. As a divergence from previous chapters, we will allow *rc* to assert even if the guard evaluates to *false*. The scheduler will not commit a rule in such a state, because *notRdy* will be asserted, and it will take the precedence over *rc*.

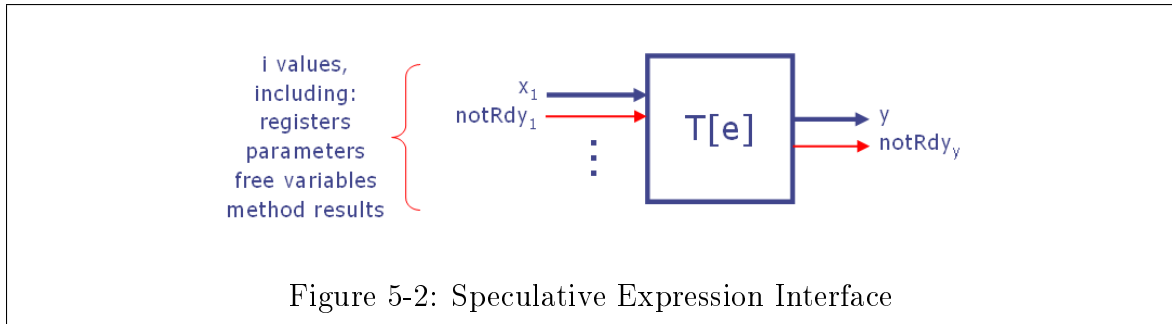
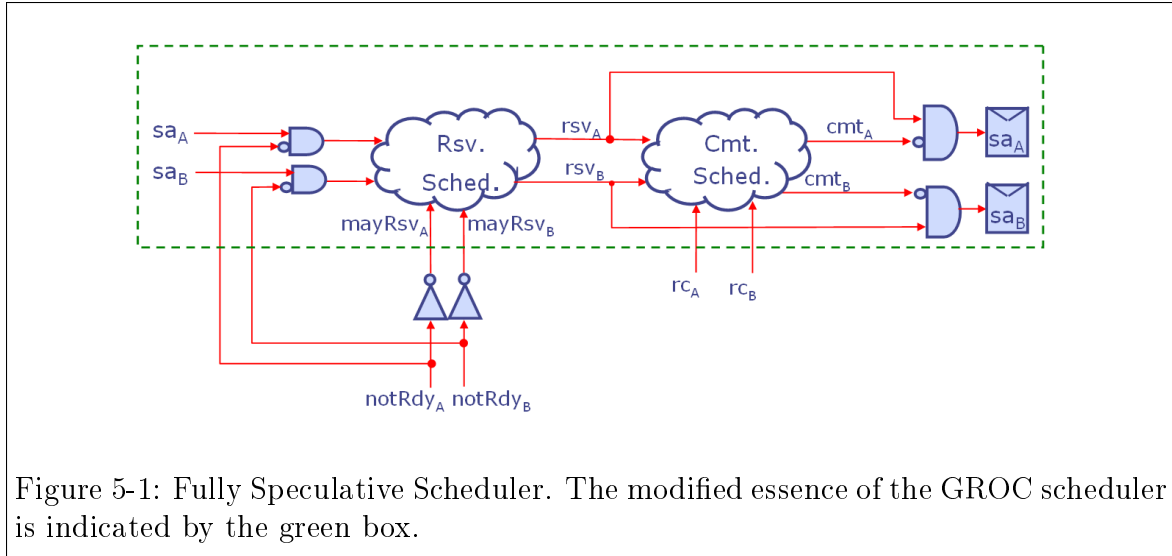
As a result of the speculative scheme above, our new scheduler will allow a rule to be reserved and released without ever being committed. The schedulers in previous chapters guaranteed that once reserved, a rule must be committed.

It is noteworthy that the use of *notRdy* matches the NR status of an expression or action in the Operational Semantics presented in Section 2.6.

Figure 6-1 presents the new speculative scheduler, which is very similar to the scheduler presented in Section 4.7. The big difference is that we are using *notRdy* rather than *rdy* signals. To accommodate this change we have to invert *notRdy* before passing it to **RsvScheduler** as *mayRsv*. We also have to remove an existing reservation for a rule that has a guard evaluating to *false*. This requires that we change the essence of the GROC scheduler, which dates back to Figure 3-5. We do this by guarding the *stillActive* bit with a corresponding *notRdy*. With those two changes, the scheduler will now reserve rules before they have evaluated their guards and release their reservations when the guard turns out to be *false*.

Once a guard evaluates to *false*, it will remain *false* until some data needed for its computation changes. At that point the guard will re-evaluate, and the rule's *notRdy* signal will be released. All this will happen automatically.

We do not have to make any changes to the setup of **CmtScheduler** because we took care of all the differences before the *rsv* signals are generated. Neither **RsvScheduler** nor **CmtScheduler** procedures have changed since they were first described in Section 3.8.

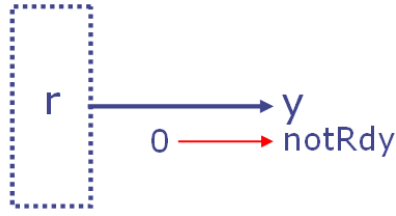


5.2 Speculative Expressions

5.2.1 Speculative Expression Interface

The addition of guarded expressions requires that we change the interface for our expressions. Guarded expressions can now produce results which are not ready. Those results can be assigned to a variable and used in another expression. This second expression must know if it is consuming values that are ready. This means that every value read in an expression must be accompanied by a corresponding *notRdy* signal. Figure 5-2

The implementation of expressions in this section will generally be similar to that from Section 4.4.



(a) Speculative Register Read Expression



(b) Speculative Constant Expression

Figure 5-3: Speculative Value Read Expressions

5.2.2 Register and Constant Expressions

Figure 5-3 presents our circuits for a basic expressions for reading the value of a register and referencing a constant. These are basically the same as those in Section 4.4.2, except that they now also output a *notRdy* signal. Registers and constant are always ready so the *notRdy* are always set to 0 here.

5.2.3 Speculative Let Expression

Figure 5-4 presents our circuit for speculative let expressions. This circuit is basically the same as that in Section 4.4.3, but also carries the *notRdy* signals between expressions.

5.2.4 Primitive Operator Expression

Figure 5-5 shows the circuit for a speculative primitive operator expression. This circuit is virtually identical to one from Figure 4-7, but includes *notRdy* signals, which are OR-ed between the two expression inputs.

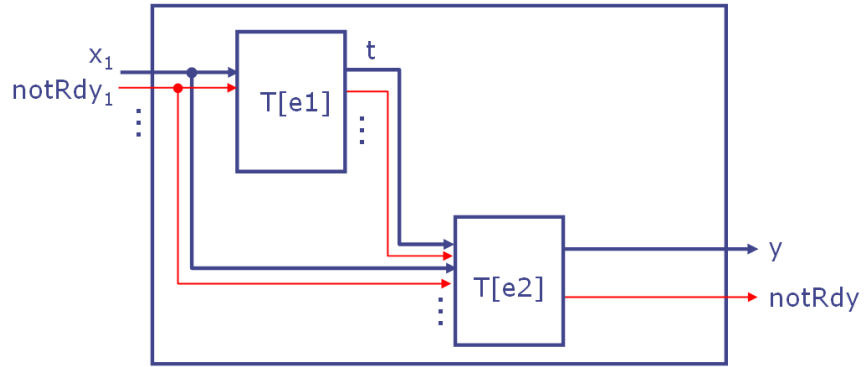


Figure 5-4: Speculative Let Expression

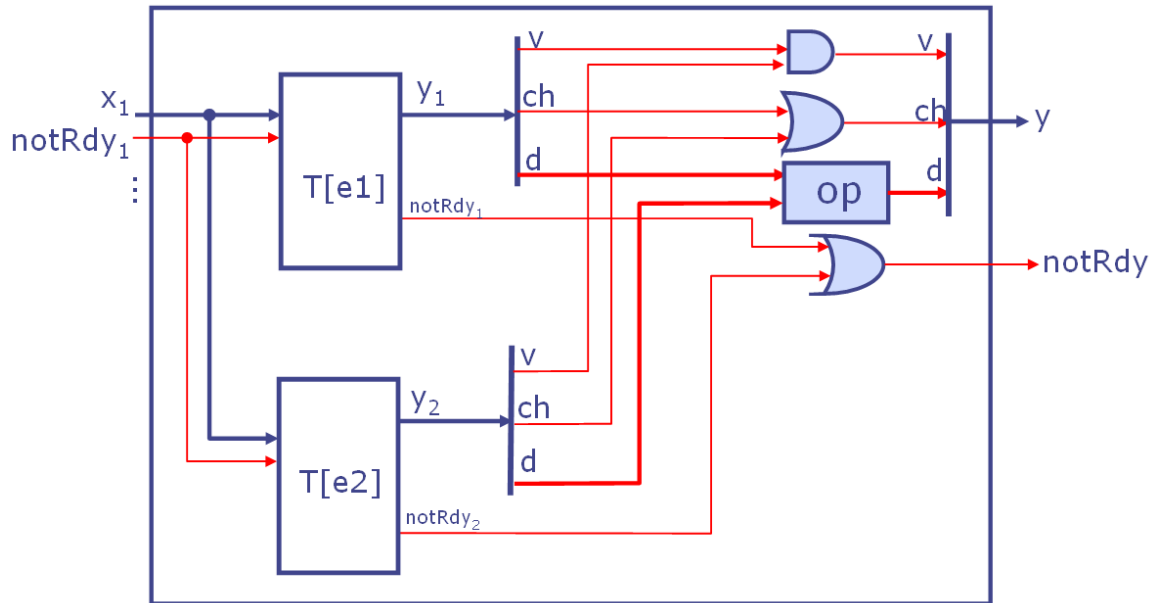
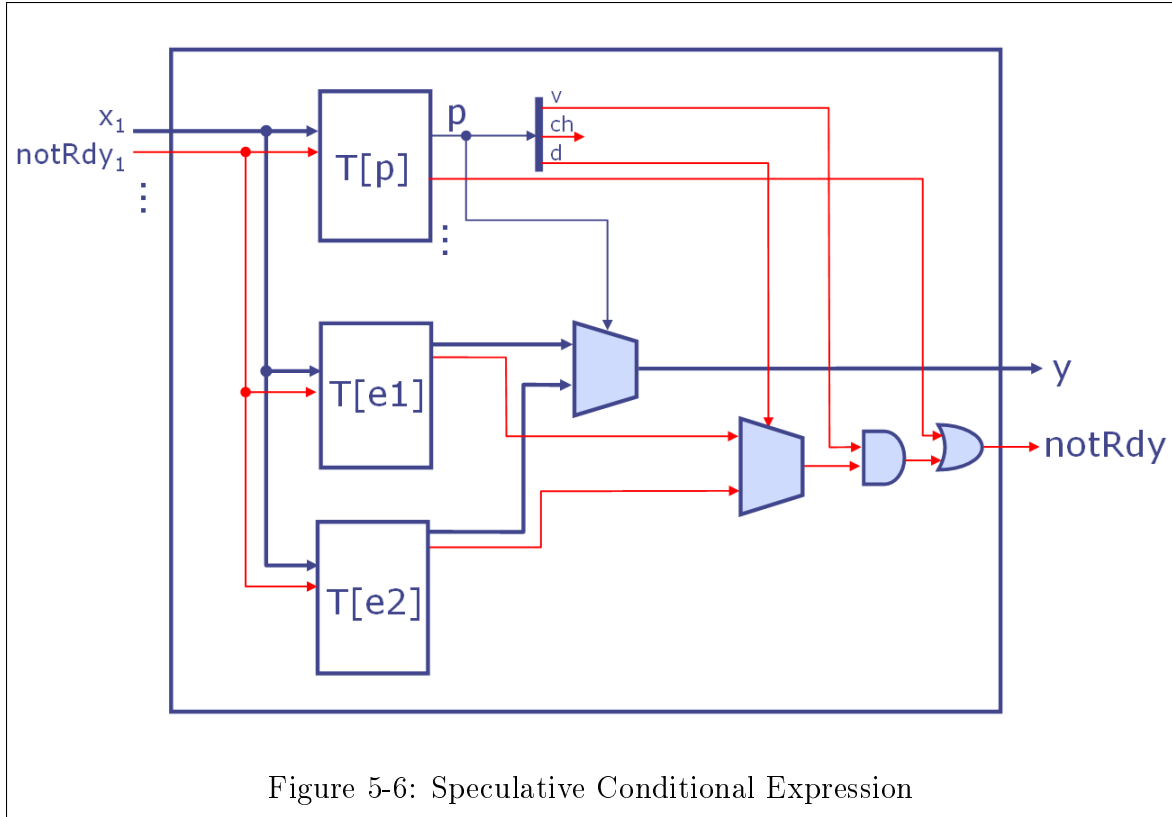


Figure 5-5: Speculative Primitive Operation Expression



5.2.5 Speculative Conditional Expression

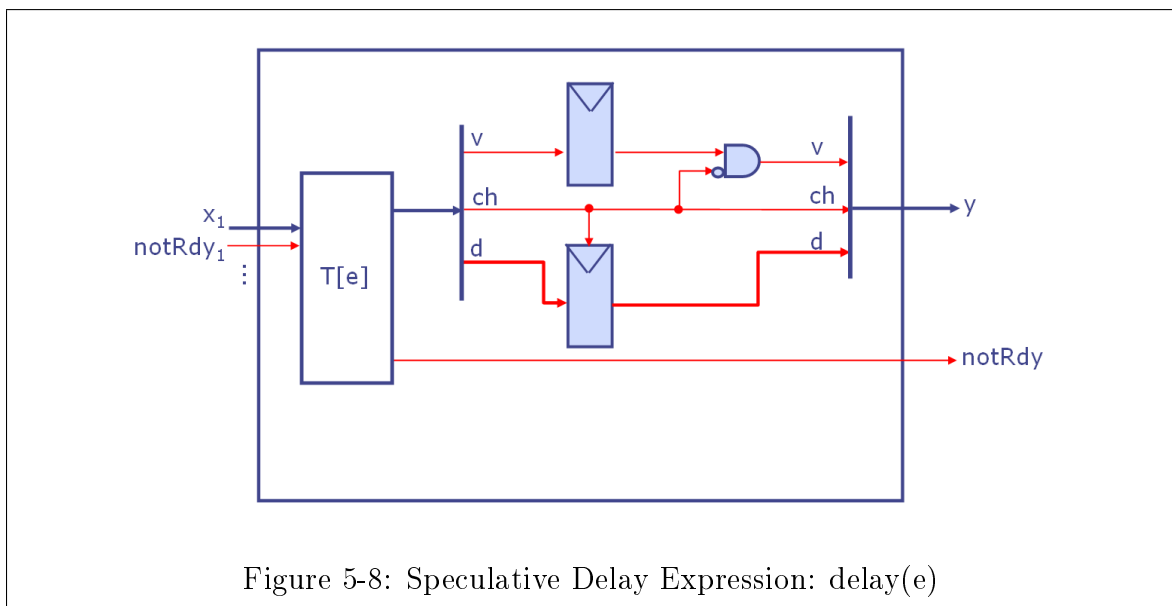
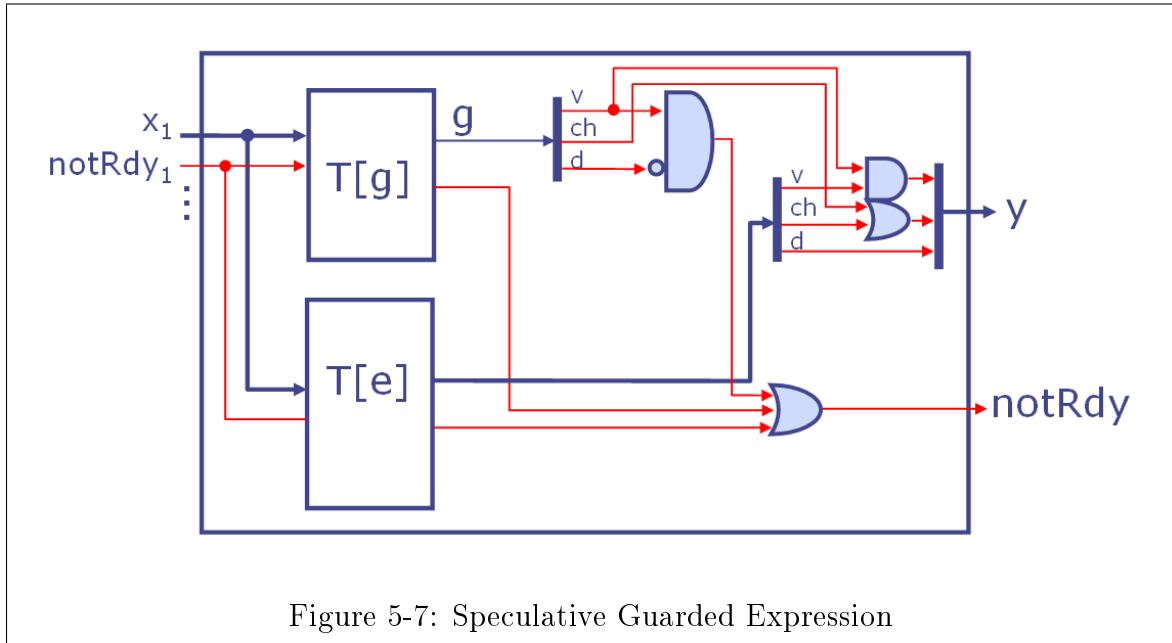
Figure 5-6 presents our new speculative conditional expression. We select the correct *notRdy* signal from *e1* and *e2* using result from *p*. The selection is only valid when *p_{valid}* is true. The result is also combined with *notRdy* from *p*. The selection of the correct result is done the same as in Figure 4-8.

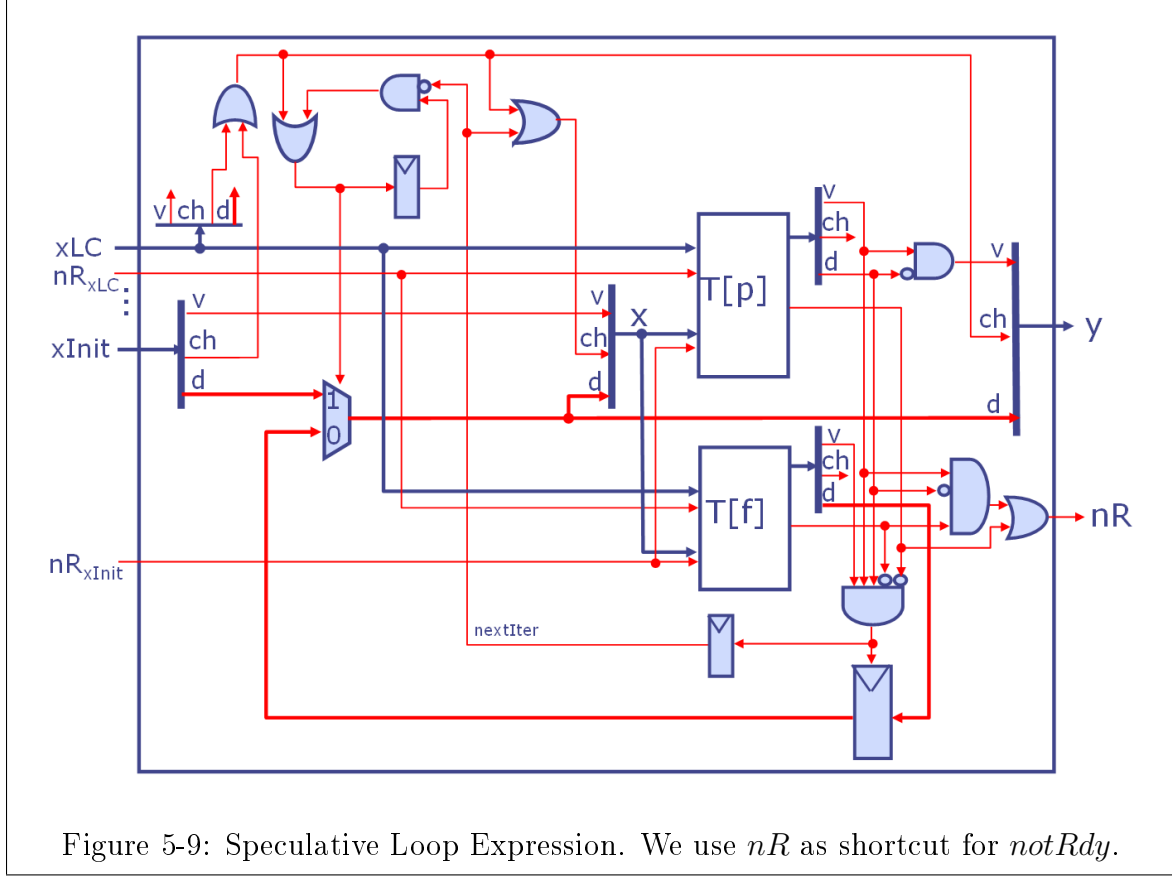
5.2.6 Speculative Guarded (When) Expression

Figure 5-7 presents our circuit for a guarded expression. The expression is ready only if both the *g* and *e* sub-expressions are ready and the guard evaluates to *true*.

5.2.7 Delay Expression

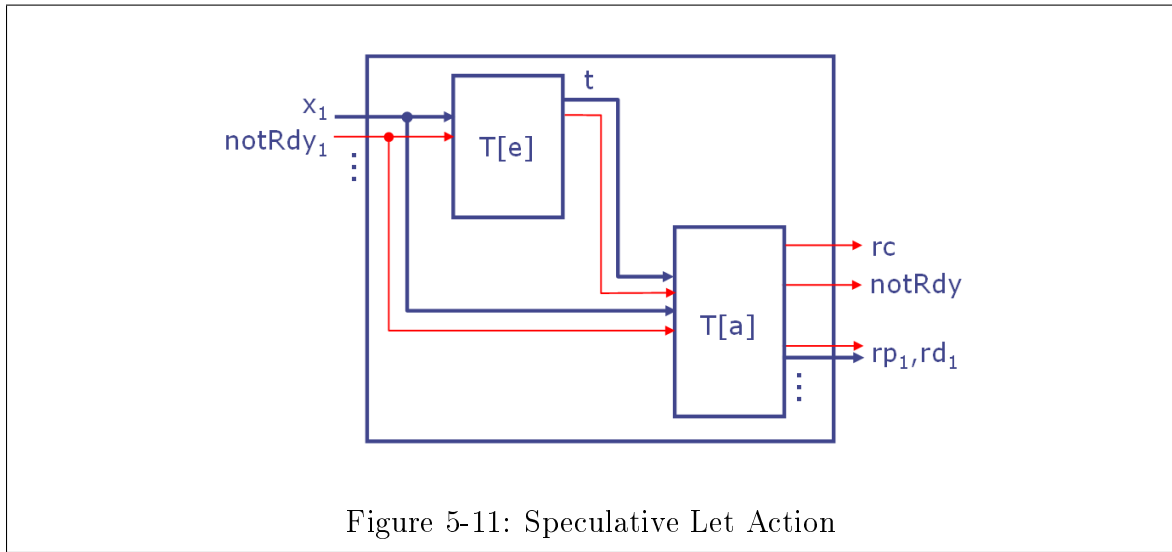
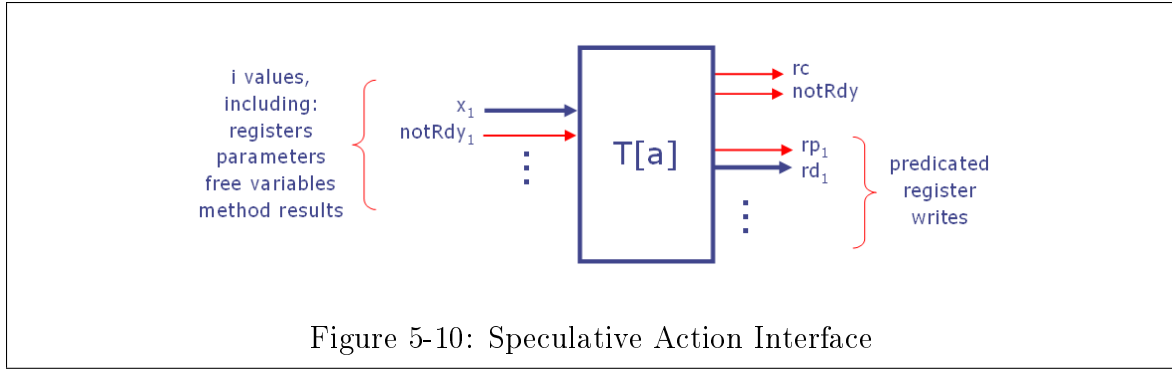
Figure 5-8 presents a speculative *delay* operator. It is virtually identical to that from Section 4.4.6. We delay the data and valid, but not *notRdy*.





5.2.8 Loop Expression

As usual, the only difference between speculative loops and non-speculative loops is that speculative loops have to carefully process the *notRdy* signals. Here the loop can be forced to stop iterating when one of the expressions (condition or body) becomes not ready. We accomplish this by including the *notRdy* signals from the loop body and condition in generation of the *nextIter* signal. If the body becomes not ready but the guard evaluates to 0, the overall expression is still ready. Figure 5-9 presents the circuit.



5.3 Speculative Actions

5.3.1 Speculative Action Interface

Figure 5-10 presents the interface for fully speculative actions. Aside from addition of the *notRdy* signals for input values, the interface is identical to that from Section 4.5.1.

5.3.2 Speculative Let Action

Figure 5-11 presents the circuit for a speculative let action. The setup is very similar to that for a speculative let expression: the output of the expression is passed into the action.

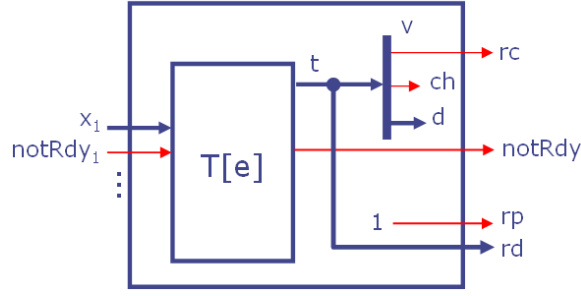


Figure 5-12: Speculative Register Assignment Action

5.3.3 Speculative Register Assignment Action

The speculative register assignment action is presented in Figure 5-12. The control wires are simply shuffled in and out of the expression. Register assignment logic is same as in Figure 4-12.

5.3.4 Speculative Conditional Action

Figure 5-13 presents our speculative conditional action circuit. The action is ready to commit when the condition is fully evaluated, and either turns out to be *false*, or the child action is ready to commit. Similarly, the conditional action is *notRdy* when the condition is *notRdy* or condition is *true* and the child action is *notRdy*. We also must guard the method and register predicates to only be asserted when the condition evaluates to *true*.

5.3.5 Speculative Guarded (When) Action

Figure 5-13 presents our guarded action circuit. The difference between the guarded and conditional actions is that when the guard evaluates to *false*, the whole action cannot be committed and we must assert the *notRdy* signal. This allows us to slightly simplify the generation of the *rc* signal by taking advantage of the fact that *notRdy* always overwrites *rc*, thus *rc* can be asserted spuriously, as long as *notRdy* is asserted as well.

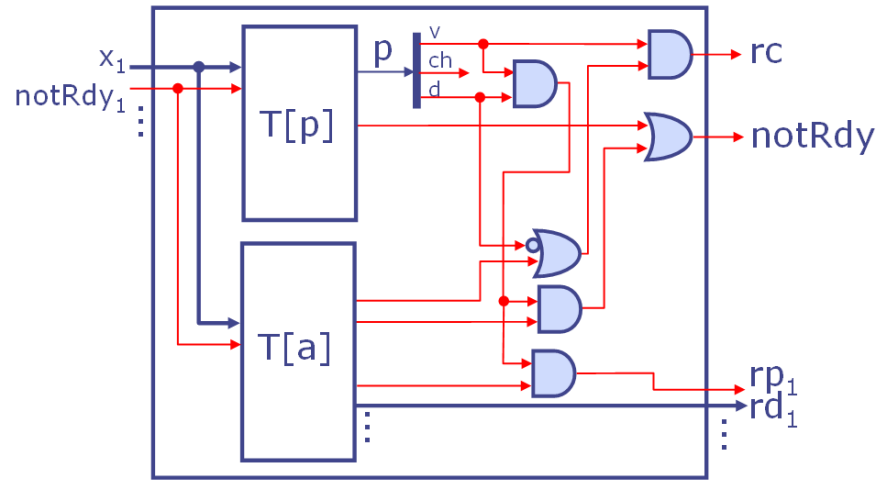


Figure 5-13: Speculative Conditional Action

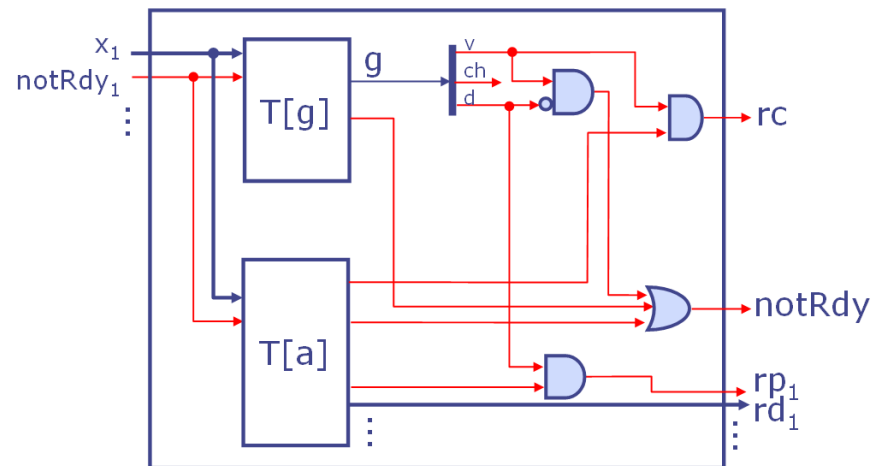


Figure 5-14: Speculative Guarded Action

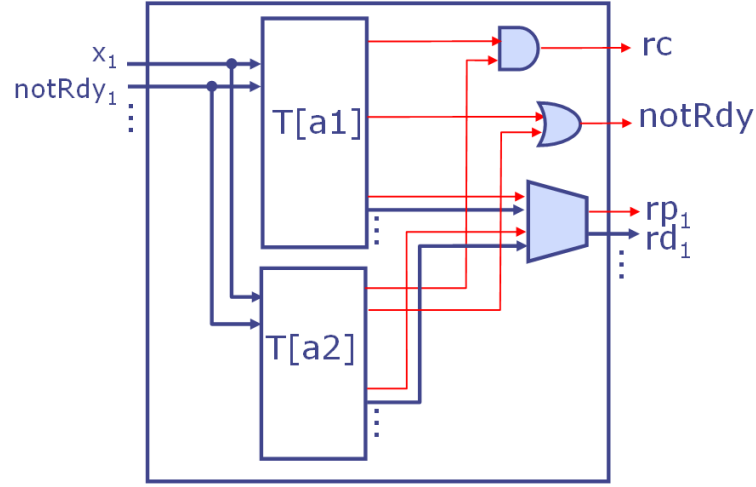


Figure 5-15: Speculative Parallel Actions

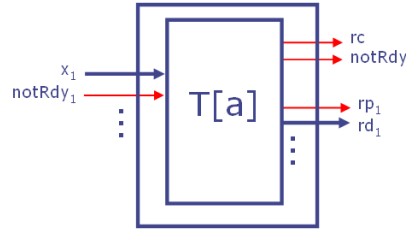


Figure 5-16: Speculative Rule Circuit

5.3.6 Speculative Parallel Action Composition

Figure 5-15 presents our parallel composition circuit. We simply combine all the control signals and select the data for method calls and register writes from the two actions.

5.4 Speculative Rules

A speculative rule is just an action. The guard for the rule has been internalized. As a result a rule has the same interface as an action (Figure 5-10). Figure 5-16 presents the trivial circuit for a speculative rule.

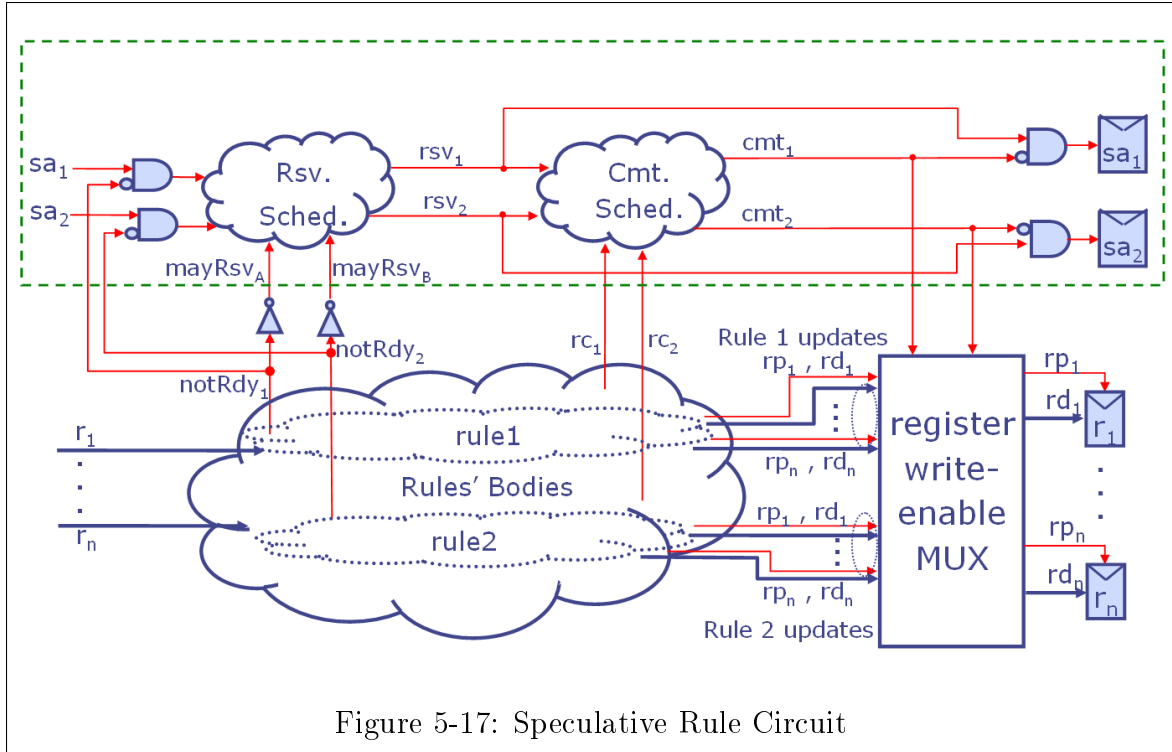


Figure 5-17: Speculative Rule Circuit

5.5 Speculative Module Model

Figure 5-17 presents the whole speculative module put together. The circuit is identical to that from Section 4.7, except that rules produce *notRdy* instead of *rdy*, and we handle it as described in Section 5.1.

Chapter 6

Hierarchical Multi-cycle Synthesis

In this chapter we will present our method for syntax-directed translation of hierarchical BMC, as described in Section 2.5. We will begin with computation of a *ReservationRequired* property for guard and data computation. We will follow this up with changes to the speculative GROC scheduler from previous chapter to take advantage of the *ReservationRequired* property. We will follow with a presentation of method interfaces, hierarchical expressions and actions and full module circuit construction. We will finish the chapter with a brief discussion of limitations of our method.

6.1 Reservation Required

Consider the method interfaces used by the Bluespec compiler described in Section 2.4. What happens when a guard needs to call a method? Bluespec compiler allows only limited access to methods from guards: only value methods allowed are those which do not accept any parameters or those which have only one caller – the guard. This means that a number of designs which are syntactically correct and logically sound cannot be compiled due to semantic restrictions. We would like to eliminate this restriction.

The reason that the Bluespec compiler cannot allow multiple calls from guards to methods with parameters is that in the Bluespec synthesis model all guards are

fully evaluated every clock cycle. Since a method port is a physical resource and can only accept one version of its parameters in any given clock cycle, only one of the guards can pass the parameters. This is a basic example of resource scarcity. Bluespec synthesis has no protocol for choosing which guard will evaluate. We would like to introduce such a protocol: in the event that a guard cannot easily complete due to resource scarcity, we will require that any rule or method accessing a scarce resource obtain a reservation before accessing the said resource. In addition to managing atomic consistency, our scheduler will become a resource arbiter as well. The Bluespec scheduler already performs this function for ports accessed in bodies of rules and methods, and avoids the need to do it for guards by restricting allowable designs. Our scheduler will remove all such restrictions and extend the arbitration to all methods accessed in rules and methods.

As we explained in Section 2.8, our hierarchical designs will only have a single, top-level scheduler. This scheduler will be responsible for scheduling rules. In order to allow all method calls in guards, we will have to determine which rules require reservation before being able to completely compute their guard. We will treat this as a static property of the rule’s guard. (That is, even if dynamically a particular rule can fully compute its guard without making a reservation in a particular clock cycle, we will still force the rule to obtain a reservation before considering the guard to be valid.) We call this property *ReservationRequired*, or *RR*. If a rule’s guard has $RR = true$, the rule must obtain a reservation before its guard is fully computed. If a rule’s guard has $RR = false$, then the rule’s guard can be fully evaluated even without reserving it.

We will assign the *RR* property not only to top-level rules’ guards, but to guards of all methods. This is because all guards can have calls which cannot be evaluated without a reservation. Furthermore, it is possible that a guard makes a call to a method which indirectly requires reservation. This means that in addition to computing *RR* for method’s guard, we also have to compute a separate *RR* for value method’s result.

Below we provide a procedure **ComputeRR** which computes the value of *RR*

property for guards of all methods and results of value methods.

Procedure ComputeRR(Module m):

1. For each module $c \in \{ \text{children of } m \}$: perform **ComputeRR**(c)
2. Create m' from m by separating guards from methods and rules and converting them to value methods. The new guard method will consist of a conjunction of all guard expressions. The new method will consist of the old method with all guards removed. m' consists of value methods and either rules or action methods.
3. Define $shared(m) = \{ \text{set of methods in } m \text{ which are called from more than one rule or method in parent of } m \}$
4. For each value method f in m' :
 - 4.1 Define $adeends(f) = \text{result of } f \text{ depends on its arguments}$
 - 4.2 Define $mdepends(f) = \{ \text{set of methods which } f \text{ uses} \}$
 - 4.3 $RR(f \in m') = (\exists g \in mdepends(f) \wedge RR(g)) \vee (f \in shared(m) \wedge adeends(f))$
5. For every rule r in m : $RR(r) = (\text{method } f \in m' \text{ corresponds to guard of } r) \wedge RR(f)$

Procedure **ComputeRR** depends on a procedure which converts our designs to equivalent designs with guards separated from bodies of rules and methods. These guards are converted to separate value methods. It then computes the RR property for all value methods in the new design, starting at the bottom of the module hierarchy. At the top-level module, the RR property of value methods generated out of rule's guards is reintegrated into the rules of the original module.

The procedure is deterministic and generates only one assignment of RR for a particular design.

6.2 Hierarchical Speculative Scheduling

In the previous chapter we have presented a speculative scheduling approach based on use of *notRdy* signals generated by the rules. Our motivation was that guards can take a long time to evaluate and we could introduce scheduling unfairness against rules with such guards if we did not provide a mechanism to evaluate the guards.

As described in the previous section, hierarchical modules provide another chal-

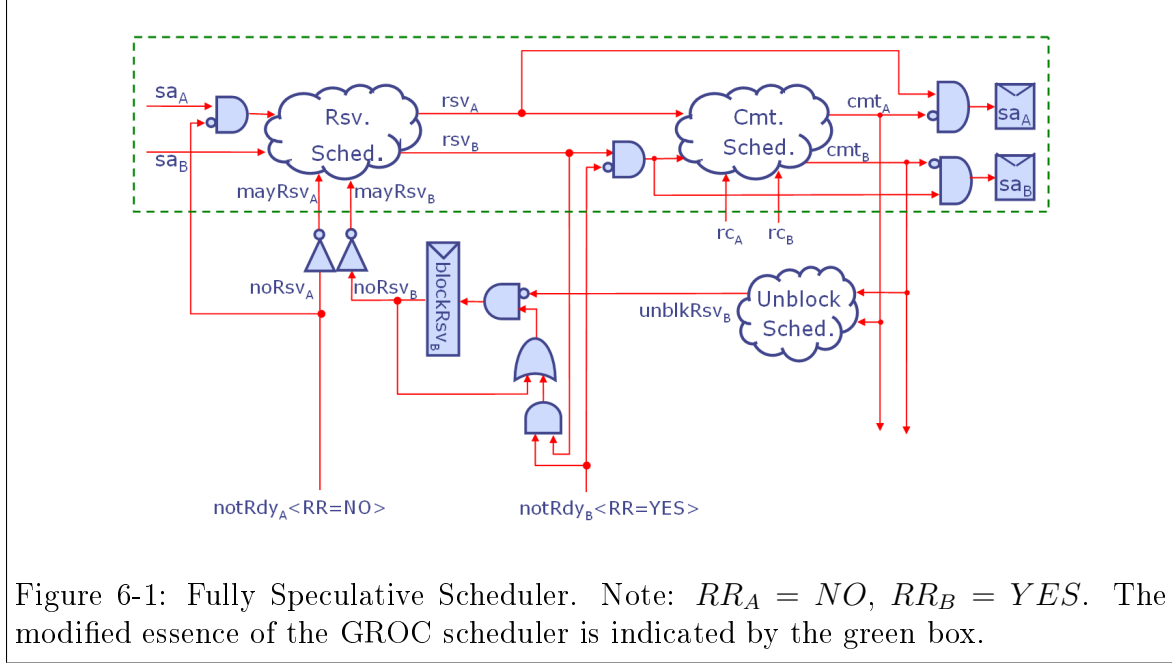
lenge for the GROC scheduling approach, because evaluating the guard may require calling a method, and calling a method may require using a scarce resource. We use reservations for managing such resources, which means that we may not be able to evaluate the guard before we reserve the rule. Rather than forbid certain features outright we would prefer to degrade them performance-wise and leave the decision to use them in the hands of the designer.

In typical designs, the guard can be fully evaluated without reserving any scarce resources. Bluespec language requires that guards not use any scarce resources. Furthermore, it is recommended that guards be simple computations that can be computed very quickly. For many rules, guards are merely implicit combinations of guards for all the methods that are called by the rule, directly and indirectly. It would be grossly inefficient to require that rules with such guards obtain a reservation before the guards are considered evaluated. Our design for hierarchical speculative scheduling avoids this problem by utilizing the *RR* property of signals described in previous section. If the rule's *RR* = *true*, then we must not consider the evaluation of the guard valid until we have reserved the corresponding rule. If the rule's *RR* = *false*, we can consider the guard's value before its rule has been reserved.

Figure 6-1 presents the new scheduler, which is a modified version of the speculative scheduler from Section 5.1. Neither **RsvScheduler** nor **CmtScheduler** procedures have changed since they were first described in Section 3.8.

notRdy_A is the signal representing the evaluation of the guard for rule *A*. Its *RR* is *false*, meaning that the guard evaluates without requiring a reservation. It is handled exactly the same way as for a flat module. Rule *B* does require a reservation in order to compute its guard fully, because its *RR* is *true*. *notRdy_B* is only valid when *rsv_B* is asserted. We cannot pass its value directly to the **RsvScheduler**, because doing so would result in a combinational loop. Instead, we remember that the guard has evaluated to *false* in *blockRsv_B* flip-flop, and pass the inverse of that value to the **RsrScheduler**.

The status of reservations is passed on to the **CmtScheduler**. If Rule *B* evaluates its guard to *false*, we remove the reservation before considering whether to commit



the rule. The updating and storing of the sa bits is the same as before.

The big divergence from the previous scheduler designs is that we have added a third phase to our scheduler. Once we know which rules are committing we use that information to determine the guards whose values may change as a result of the commits. Any rule which has its guard updated is unblocked from receiving reservations. In our example below that would be rule B . Rule A does not need to be unblocked because it never becomes blocked, because its guard does not need a reservation to be fully evaluated.

Note that the unblocking mechanism used here is rather optimistic. If the guard expression is $x \wedge m.f()$, with $x = false$ and we update data contributing to computation of $m.f()$, we will reconsider the encompassing rule for scheduling. This is even though the guard is guaranteed to continue evaluating to $false$. A more fine-grained approach would result in fewer spurious reservations of rules. In interest of clarity, our system skips this optimization. In either case, the general guidelines for guard writing is that they should be easy and fast to evaluate, so the optimization above may not be necessary, and in fact it may negatively affect the performance of the generated circuits due to additional circuitry and associated delays.

6.3 Hierarchical Speculative Method Interfaces

The method interfaces described in Section 2.4 work well for single-cycle scheduling. Multi-cycle scheduling presents some challenges that are not present in the single-cycle case:

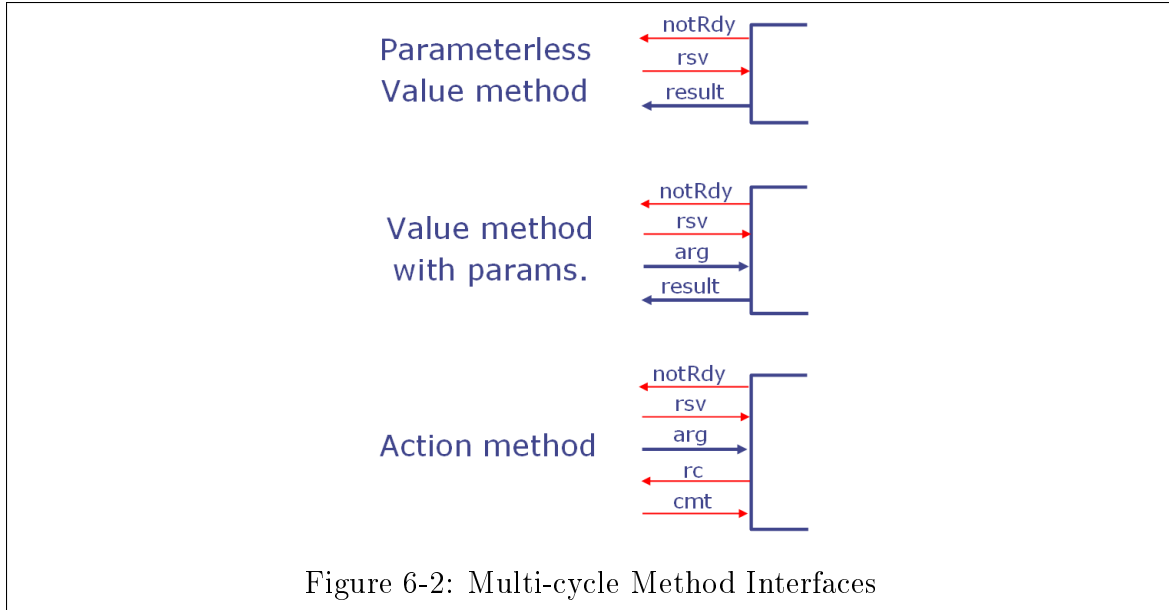
- data on wires may not be valid or may be changing,
- a call may last multiple clock cycles, so we must introduce the concept of call reservation and call completion, and
- an action method call may be guarded by an expression which cannot be fully evaluated before starting the call (evaluation needs reservation), which means we must be able to reserve an action method and release it without ever committing its updates.

We have solved the first problem in previous chapters. We introduced 3-wire logic in order to deal with exactly these types of problems. We will extend the interface data wires to consist of 3-wire logic. This will also lead to an interface more consistent with the data usage we have inside our actions and expressions.

Similarly, we have already solved the problems with the last two challenges as well. Our schedulers have more than just the *rdy* and *enable* signals provided by the single-cycle scheduler. Our control signals are:

- *notRdy* (not ready)
- *rsv* (reserve)
- *rc* (ready to commit)
- *cmt* (commit)

We will use exactly these signals to construct our method interfaces. Figure 6-2 presents the interfaces used by multi-cycle methods. We will discuss the protocol of calling a method in the next section.



6.4 Speculative Method Call Protocol

The signaling observed at the method interface is as follows:

- A method is reserved (called) by asserting its *rsv* signal. The *rsv* signal must remain asserted throughout the entire call. The caller indicates the end of the reservation by deasserting the *rsv* signal.
- Parameters are computed and passed to the method.
- An action method can signal that it is ready to commit by asserting the *rc* signal. Once *rc* is asserted it must remain asserted until the reservation is released, possibly after also being committed.
- If the method indicates that it is ready to commit, the caller may commit it by asserting the *cmt* signal, and removing the *rsv* signal in the next clock cycle. This applies for action methods only.
- The method can signal that its guard evaluates to false by asserting the *notRdy* signal. The caller must eventually end the call by removing the *rsv* signal. It is possible and allowed that a method assert *notRdy* and *rc* at the same time. In

such an event, the *notRdy* signal takes precedence. Thus we must ensure that *notRdy* is computed no later than *rc*.

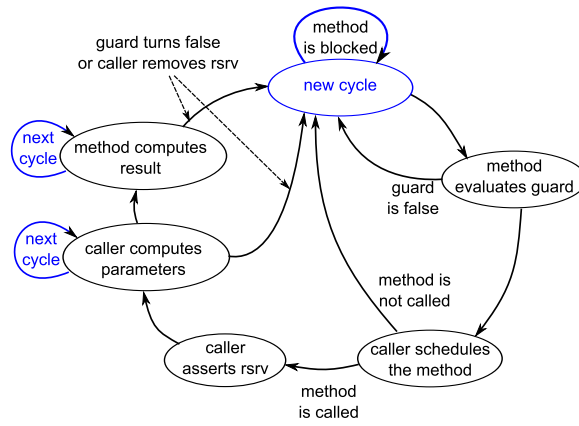
- The caller can release the reservation at any time without committing an action method or waiting for the result of a value method.

Figure 6-3 presents the above call protocol for our speculative method interface. We separated out the protocol for calling a value method and an action method. This is to draw a distinction between methods that need to be committed (action methods) and those that do not (value methods).

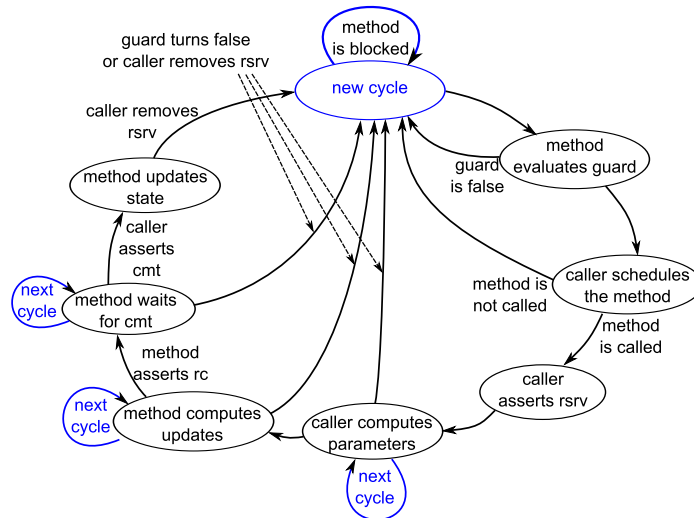
The common thread between the two protocols is the need to reserve a method, possibly speculatively, and the method's ability to inform the caller that the method cannot successfully complete its computation. There can be two reasons for a method to be unable to complete its computation: the method has a guard (from a guarded action or expression) which evaluates to *false*. Alternatively another method, lower in the call hierarchy, has a guard that evaluates to *false*. Neither of these situations can be always detected without a reservation, because the evaluation of these guards may require reserving a shared value method with parameters. Reservation decisions are made at the top-level module for rules and propagated downwards, so a method may not be able to evaluate its guard without the encompassing rule receiving a reservation.

It is vital that as soon as a method call begins, the method being called immediately reserves all the resources it needs to complete its computation. Once reserved (with a *rsv* signal), the method immediately (in the same clock cycle) reserves all the resources it needs to complete, and keeps them reserved until the method is released. This will include other methods called by this method. Methods release their resources immediately once their *rsv* signal is removed (also in the same clock cycle). Action methods commit their updates when their *cmt* signal is asserted (in the same clock cycle).

Once a method indicates that its guard evaluates to *false*, that information bubbles up to the top-level rule. An action method with a *false* guard cannot be



(a) Value Method Call Protocol



(b) Action Method Call Protocol

Figure 6-3: Multi-cycle Speculative Method Call Protocol

committed. It is possible that the top-level rule can still commit, because the guard is obscured by a conditional which will prevent the method from being committed. If it turns out that the top-level rule cannot commit, it will remove the reservation signal without committing, and the scheduler will mark the rule as blocked.

Note that the immediate reservation of the entire method call tree requires that if two rules or methods use a single method port with parameters, we cannot reserve the second rule or method if the first one is active, unless we can guarantee that at most one of them will need that method port. If it cannot be safely determined dynamically we must statically mark the two rules or methods as conflicting.

6.5 Method Port Sharing and Multiplexing

Method ports are considered to be scarce because each method exposes only one port. In our execution scheme, it is possible, in a single clock cycle, to activate two rules or methods that both contain calls to the same method, as long as we can statically prove that dynamically only one of them will actually call it. If the compiler is unable to prove that only one of the calls can be active during any clock cycle, it returns a compilation error. We call this single-cycle sharing, to distinguish with sharing the port on different cycles. We can always allow two rules or methods to share a port by declaring them to be conflicting and only reserving one of them at a time.

It is also possible for a single method or rule to internally have multiple uses (call sites) of a single scarce resource method port, as long as we can prove that only one of those uses will need to be active in a particular activation of the encompassing rule. We call this “multiplexing”.

In the case of flat modules, we did not have to explicitly consider resource sharing as part of the scheduling process. The only scarce resources rules were competing for were register write ports. Since we were explicitly searching for an equivalent of a sequential schedule, we were able to resolve that resource contention fairly easily by committing only the logically latest register updates. Now that we have introduced hierarchy, we must treat method ports as resources which may be scarce.

There are three types of method ports we will consider:

- value method ports without parameters,
- value method ports with parameters, and
- action method ports.

Ports of value methods without parameters can be shared and multiplexed freely by callers (rules and methods) without concern for resource contention. This is because at any point in time all callers to a value method without parameters must observe the exact same result.

Ports of value methods with parameters can not be shared in a single clock cycle. This is because the parameters may have an influence on the result of the call, and a single port can only accept a single set of parameters and return a single value.

Ports of action methods cannot be shared in a single clock cycle either. An action method must commit its updates, and it is difficult to create an interface that will allow an action method to reliably commit its updates twice in a single clock cycle. While it is possible that for some action methods it is legal to only pass on the last call (similar to the case of register writes), we consider this an optimization requiring little more than additional book keeping, and leave for future work.

Ports of value methods with parameters and action methods can always be multiplexed, provided that we can guarantee that only a single call site within the caller is actually using the port. We must be able to verify this guarantee statically. Failure of such verification means that the design may be invalid.

We enforce the rules about method port sharing by adding them to our computation of conflicts between rules and methods: in addition to the rules given previously, we will also consider any two rules or methods accessing the same scarce method port as conflicting.

In order to support port sharing and multiplexing, we will have to pair method parameters with a method predicate (mp) to create a (mp, md) tuple. This method predicate holds a similar function to the register predicate we used in Section 4.5.1.

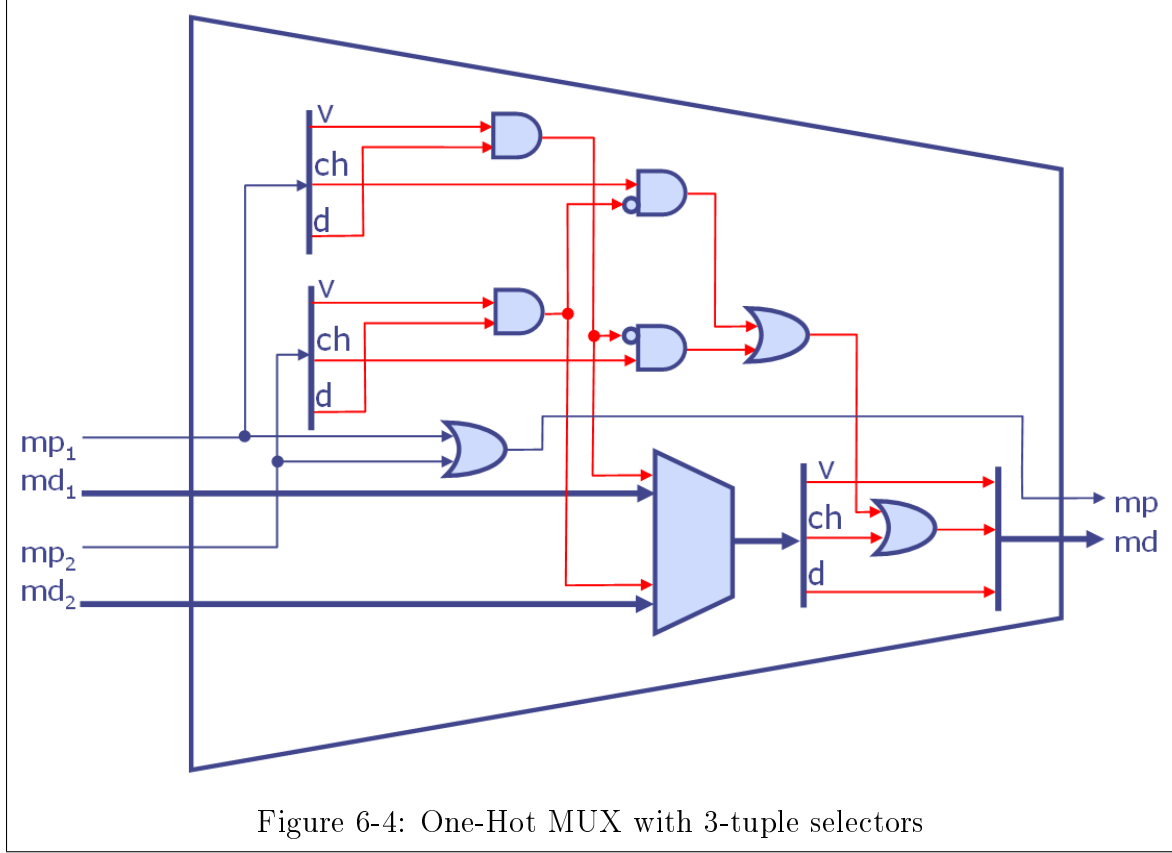
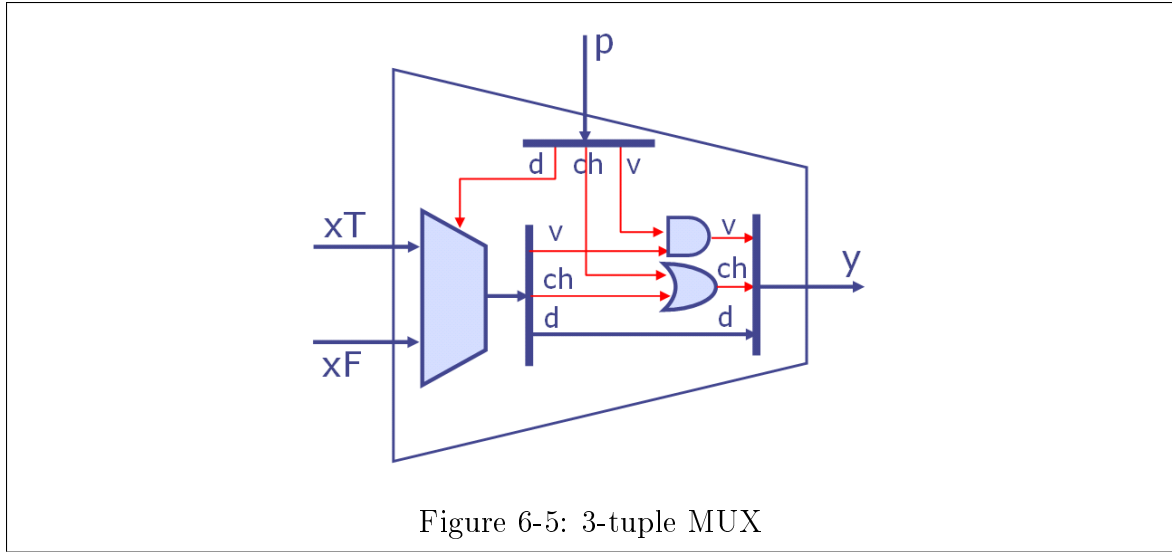


Figure 6-4: One-Hot MUX with 3-tuple selectors

The method predicates are 3-tuple logic, because we need to correctly compute the *changed* signal of the method parameters. We will use the method predicate wires to merge method calls for multiplexing because we need to merge calls that are “far away” from each other but are statically provable to be disjoint (ex. $(x > 0 ? m.f(y) : y) + (x < 0 ? m.f(z) : z)$). Port sharing by definition has the calls “far away” from each other (different rules or methods) so it also requires method predicates. In order to support use of these method predicates we introduce a one-hot MUX with 3-tuple selectors in Figure 6-4. Also, in order to support port multiplexing across conditional expressions, we introduce a 3-tuple MUX in Figure 6-5. 3-tuple MUX will allow allow us to take the wire selecting a branch of a conditional action or expression and chose which value of the parameter we chose to pass on. 3-tuple MUX is basically the conditional expression circuit from Figure 4-8.

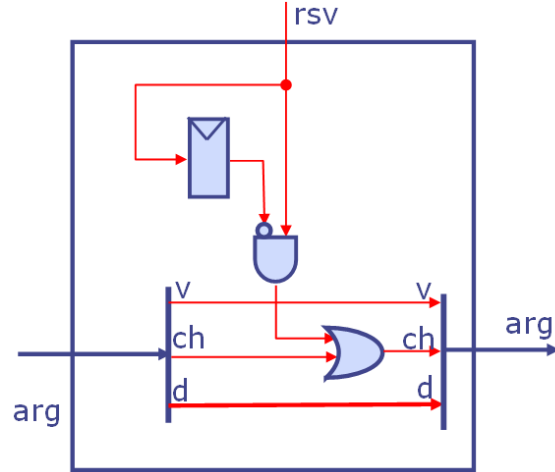
We must also consider how the method predicates interact with reservation signals



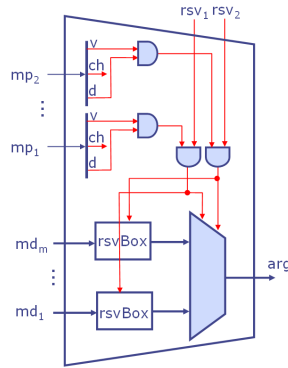
of the caller methods or rules. The problem here is that it is possible for the reservation signals to reserve a different method or rule to use a particular port without the underlying data from any method or rule changing. But such a change in the reservation signal does mean that the parameter will change and we must assert the *changed* wire in the 3-tuple parameter. We must also combine the reservation signal with method predicates. To accomplish all this, we introduce **rsvBox** and **argBox** circuits in Figure 6-6, which together will assert the changed wire correctly.

The first component (**rsvBox**) of the solution is duplicated for every user of a shared port. **rsvBox** remembers if that caller was active in the previous clock cycle. When the caller is reserved in current clock cycle but has not been reserved in previous clock cycle, **rsvBox** asserts the *changed* wire of the argument. The second component (**argBox**) combines the corrected arguments and passes them through a regular one-hot MUX. The selector of the MUX (and input to the **rsvBox**) is the *rsv* signal combined with the method predicate. The result can now be passed on to the method port.

Note that we only need to use **argBox** circuits when a method with parameters has more than one caller rule or method. If a method has only one caller, the parameter is passed directly from the caller to the method without the need for selecting the value. Thus we do not need to modify the changed wire of the parameter.

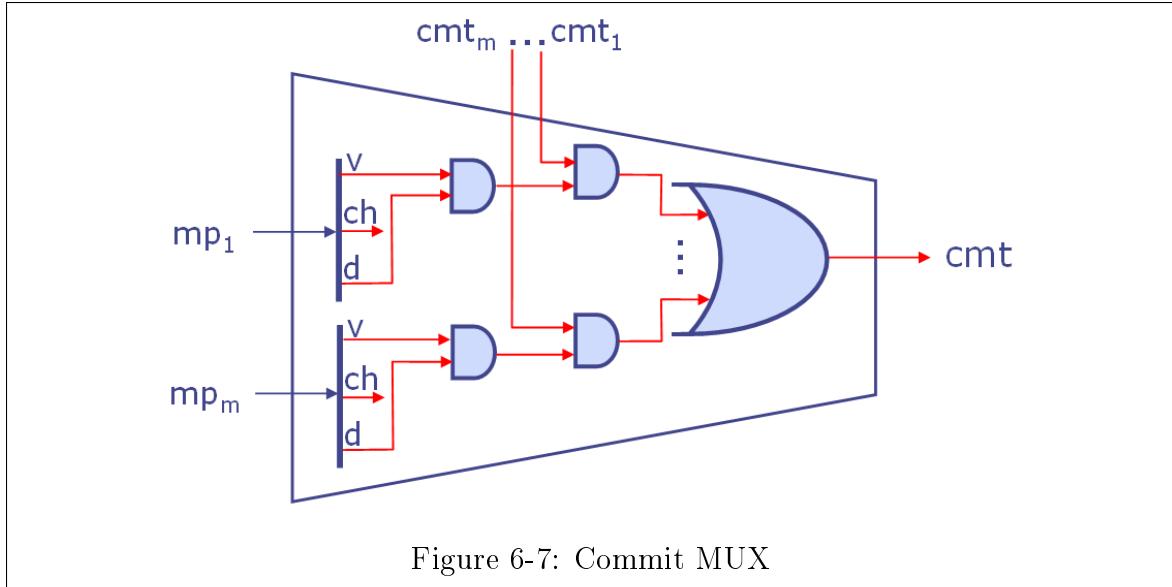


(a) **rsvBox** - asserts arg_{ch} in first clock cycle rsv is asserted



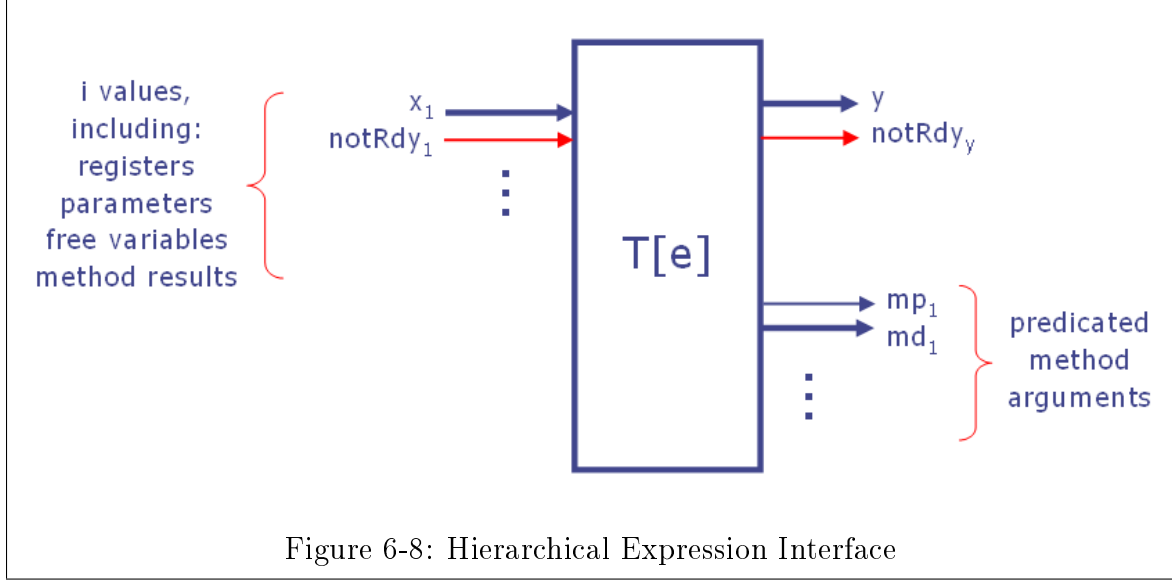
(b) **argBox** - uses **rsvBox**'es and MUXes the corrected data

Figure 6-6: Argument MUX



As we mentioned above, an action method can occasionally be called but not committed. To implement this, we must prevent the *cmt* signal on the method port from being asserted when the caller is committed. We have already developed a mechanism to achieve this: the method predicate (*mp*) wires can be used to determine whether a particular caller needs to commit the action method. We will combine the method predicate with the commit signal for all potential callers to generate the final commit for the method. This means that we must propagate the method predicates for all action method calls, not just those with parameters.

The circuit for this computation, **cmtMux**, is presented in Figure 6-7. The **cmtMux** will only assert the output *cmt* in clock cycle *t*, if there is a caller to the method which is being committed in clock cycle *t*, and which has evaluated its method predicate to true. We must insert a **cmtMux** for every action method port, regardless of whether it is shared or not.



6.6 Hierarchical Expressions

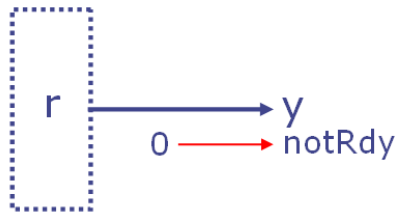
6.6.1 Hierarchical Expression Interface

The change to our language to be hierarchical requires a change to the smallest of our circuits. One of the expressions available is a value method call, which produces a (mp, md) tuple. We will have to compose those signals in addition to managing all the signals for speculative expressions introduced in Section 5.2.

The expression interface is presented in Figure 6-8.

6.6.2 Hierarchical Value Expressions

Figure 6-9 presents our circuits for expressions reading the value of a register and a value of an argument. Both of these are always ready. While this seems obvious for reading registers, it is less so for reading arguments, because if the method is not reserved then the argument may not be valid. We do not have to worry about the situation where the argument is not valid, because in such a situation, the output of the expression will have the RR property set to *true*, meaning that we will not make any decisions based on this value until we obtain a reservation for the parent rule.



(a) Hierarchical Register Read Expression



(b) Hierarchical Argument Read Expression

Figure 6-9: Hierarchical Value Read Expressions

6.6.3 Hierarchical Let Expression

Figure 6-10 presents our circuit for let expressions. For the most part, we simply evaluate the two expressions, using the result of the first one in the evaluation of the second one. Since both expressions may have calls to the same value methods with parameters, we must pass those matching pairs through a one-hot MUX. As mentioned before, our compiler must be able to verify that the two calls are exclusive. If not, the program may be invalid.

6.6.4 Hierarchical Primitive Operator Expression

Figure 6-11 shows the circuit for a hierarchical primitive operator expression. This circuit is virtually identical to one from Figure 5-5, but includes method parameters which pass through the one-hot MUX.

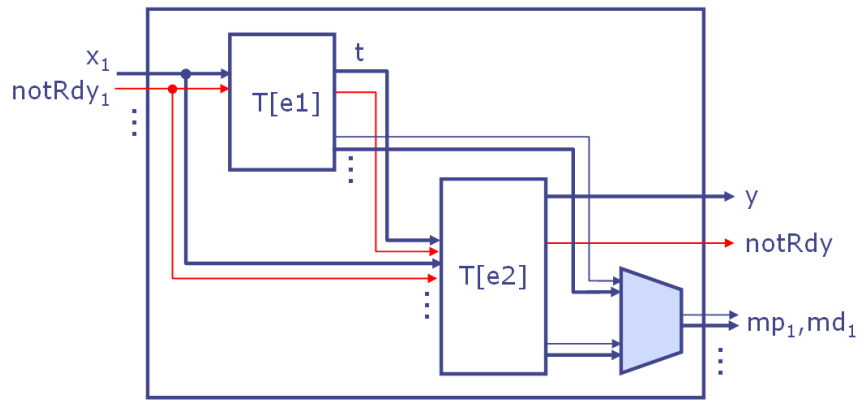


Figure 6-10: Hierarchical Let Expression

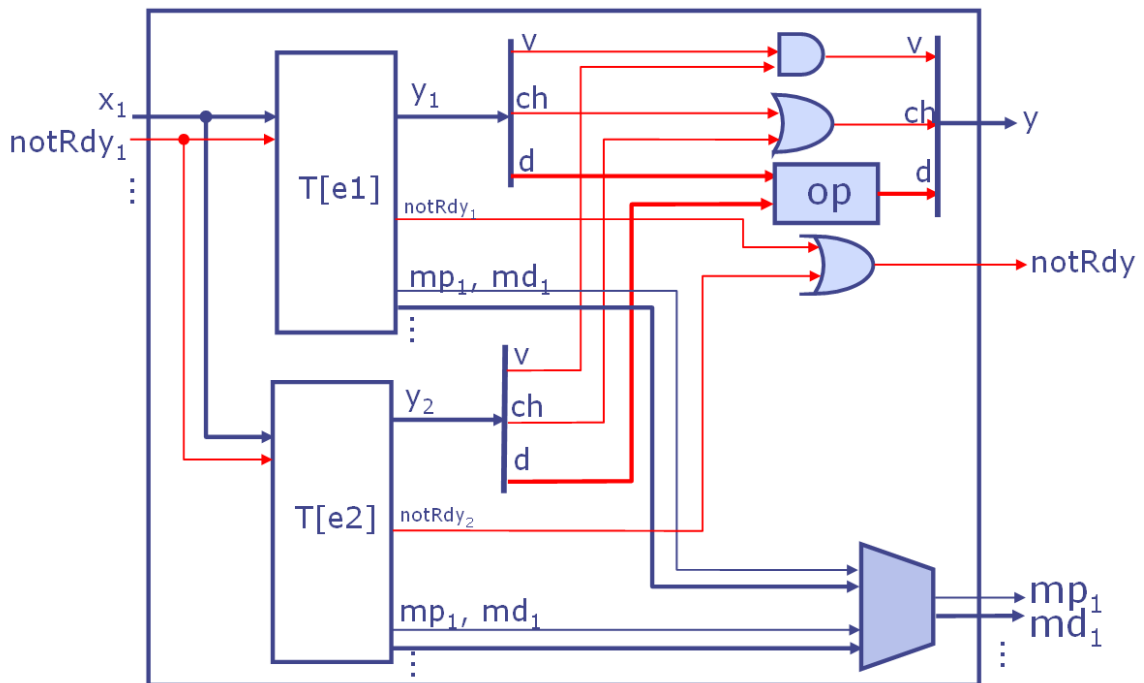


Figure 6-11: Hierarchical Primitive Operation Expression

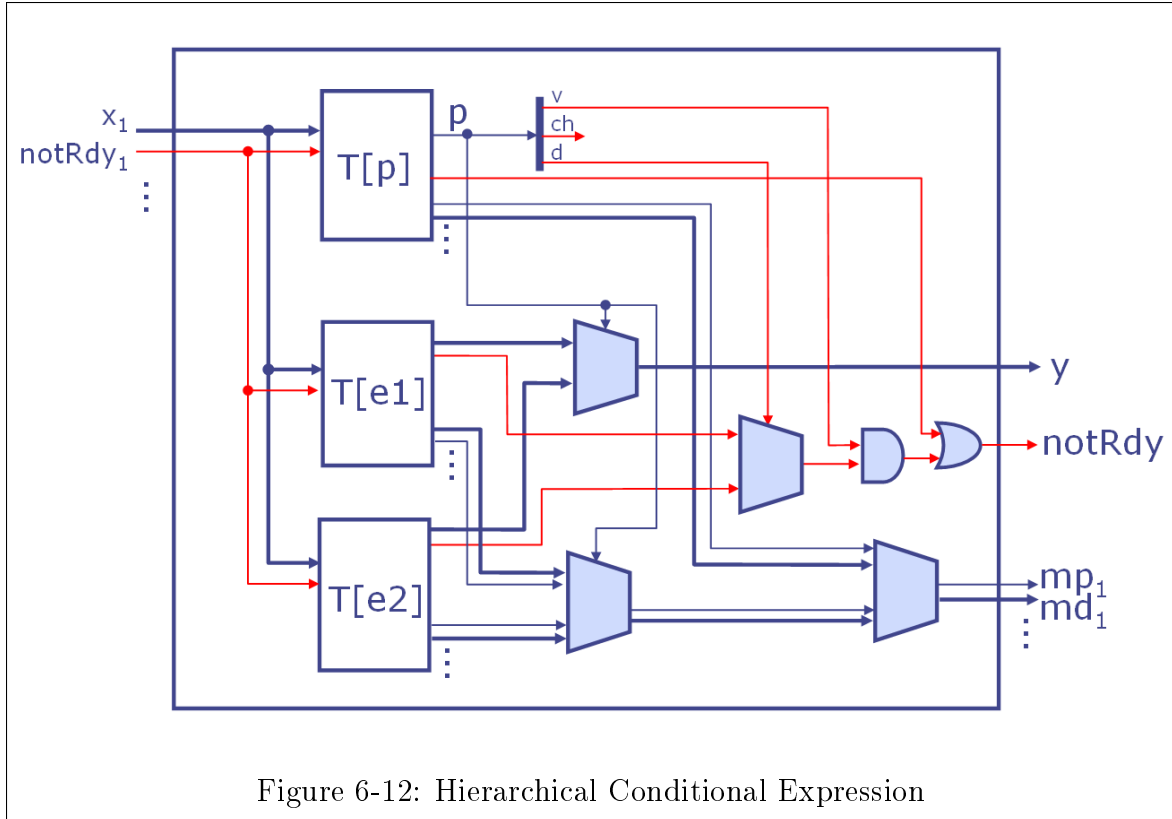


Figure 6-12: Hierarchical Conditional Expression

6.6.5 Hierarchical Conditional Expressions

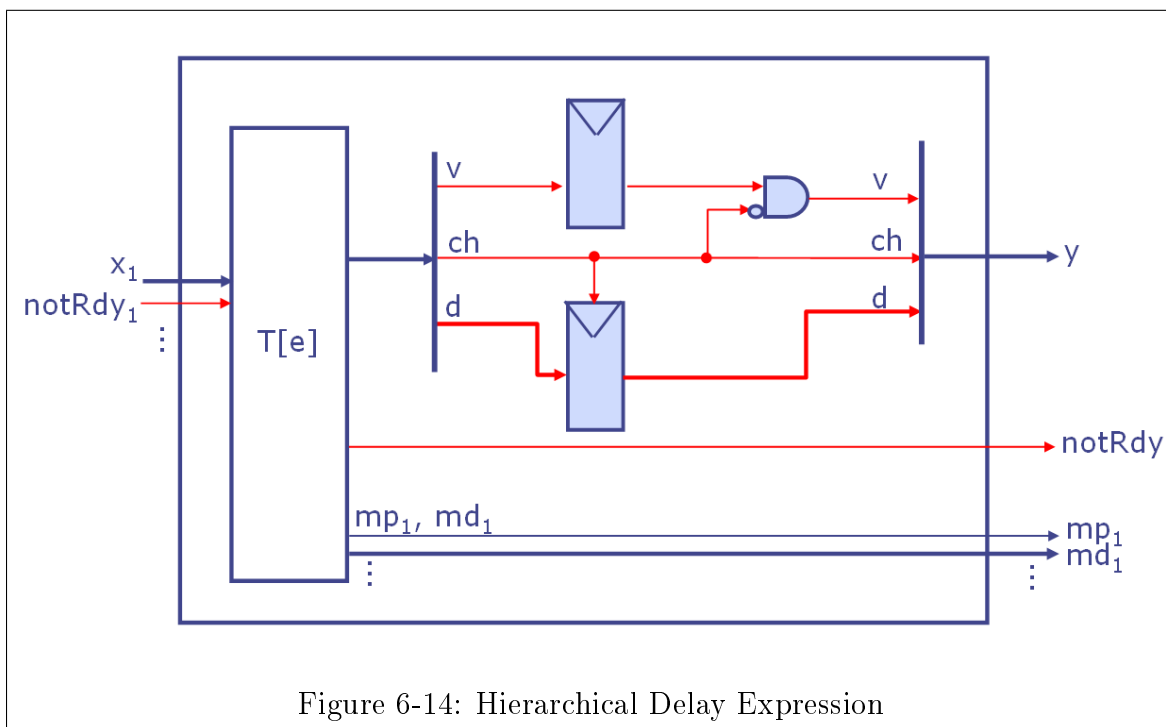
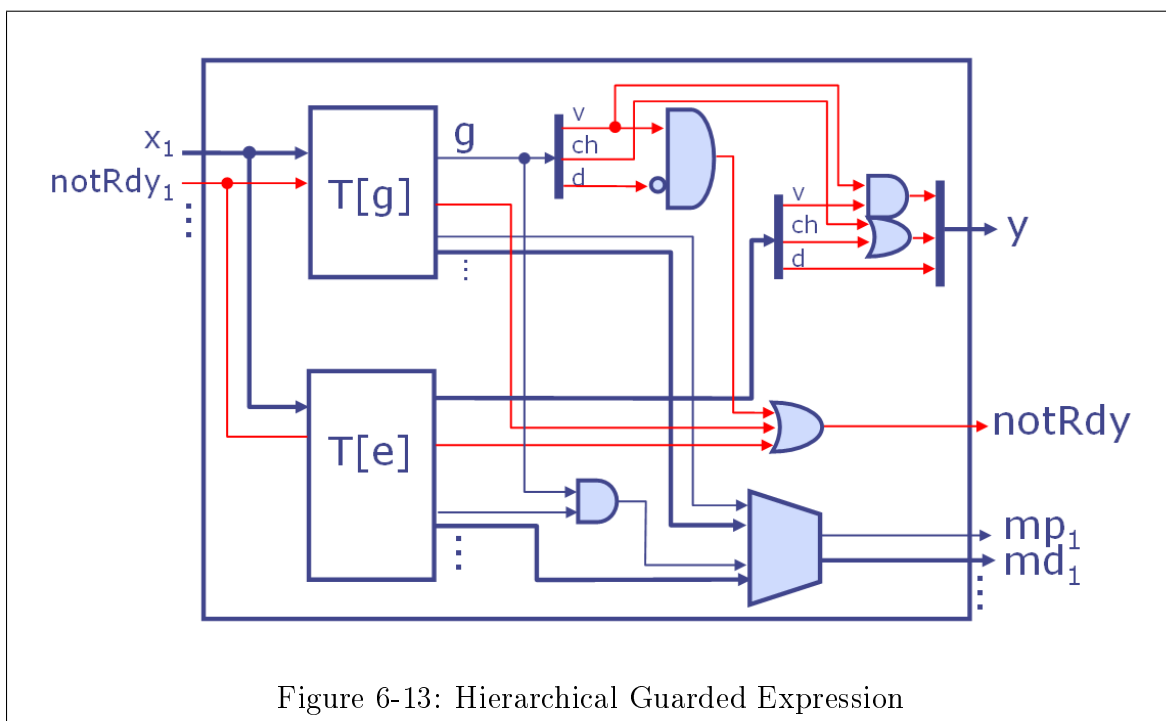
Figure 6-12 presents our hierarchical conditional expression which is very similar to that from Figure 5.2.5. We select the correct method data and predicates from $e1$ and $e2$ using p , and combine them with method data and predicates from p . The computation of the result and notRdy is done the same as in Figure 5-6.

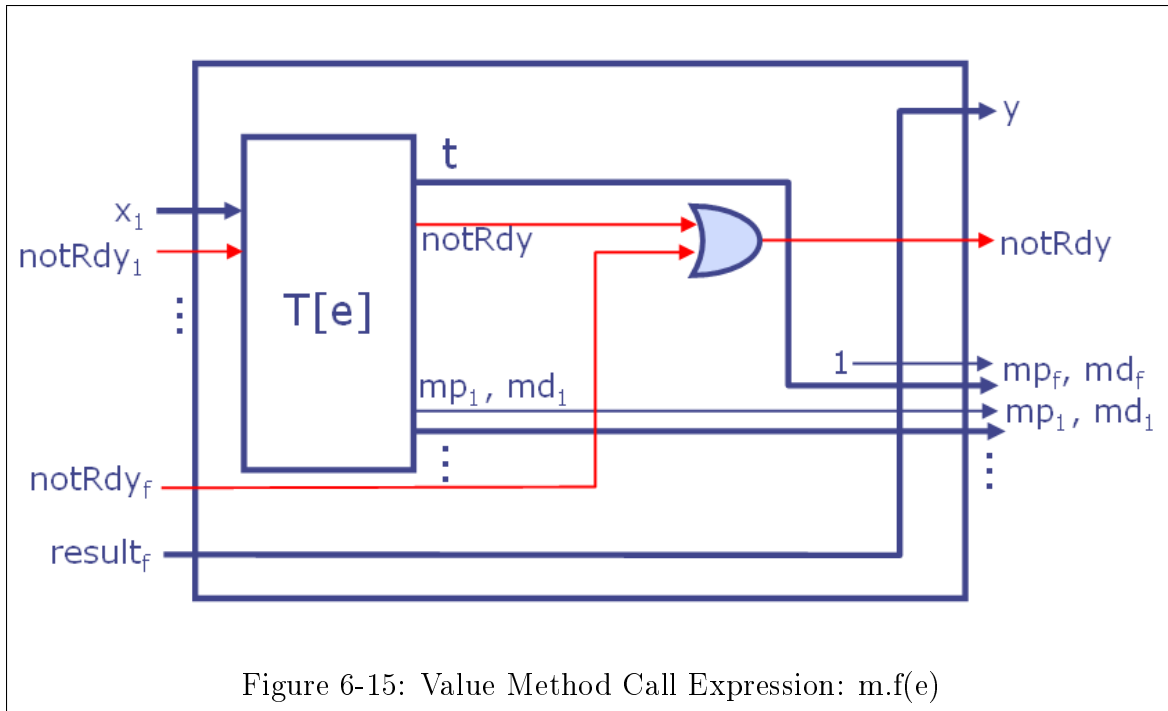
6.6.6 Hierarchical Guarded (When) Expression

Figure 6-13 presents our circuit for a guarded expression. The expression is the same that in Figure 5-7, but also includes the method predicate and data signals.

6.6.7 Hierarchical Delay Expression

Figure 6-14 presents a hierarchical delay operator. It is virtually identical to that from Section 5-8. We delay the data and valid, but not any other signals.





6.6.8 Value Method Call Expression

Figure 6-15 presents the circuit for a value method call expression. We separated out the result and *notRdy* from the value method from the inputs to the expression computing the arguments to the method. The *notRdy* signals from the method and from the arguments are combined together, because we cannot make the method call without being able to compute the arguments. Note that the operational semantics of a method call specifically disallow passing arguments which are NR.

6.6.9 Loop Expression

The syntax for loops in our language specifically excludes any hierarchical constructs from the loop expression or condition. This means that syntactically and semantically loops in our hierarchical language do not differ in any way from the loops in our speculative language. The description and figure in Section 5.2.8 applies to loops in hierarchical compilation as well.

The reason that we must keep our loops flat is that hierarchical loops provide an additional challenge, which has not been observed in other expressions: as part

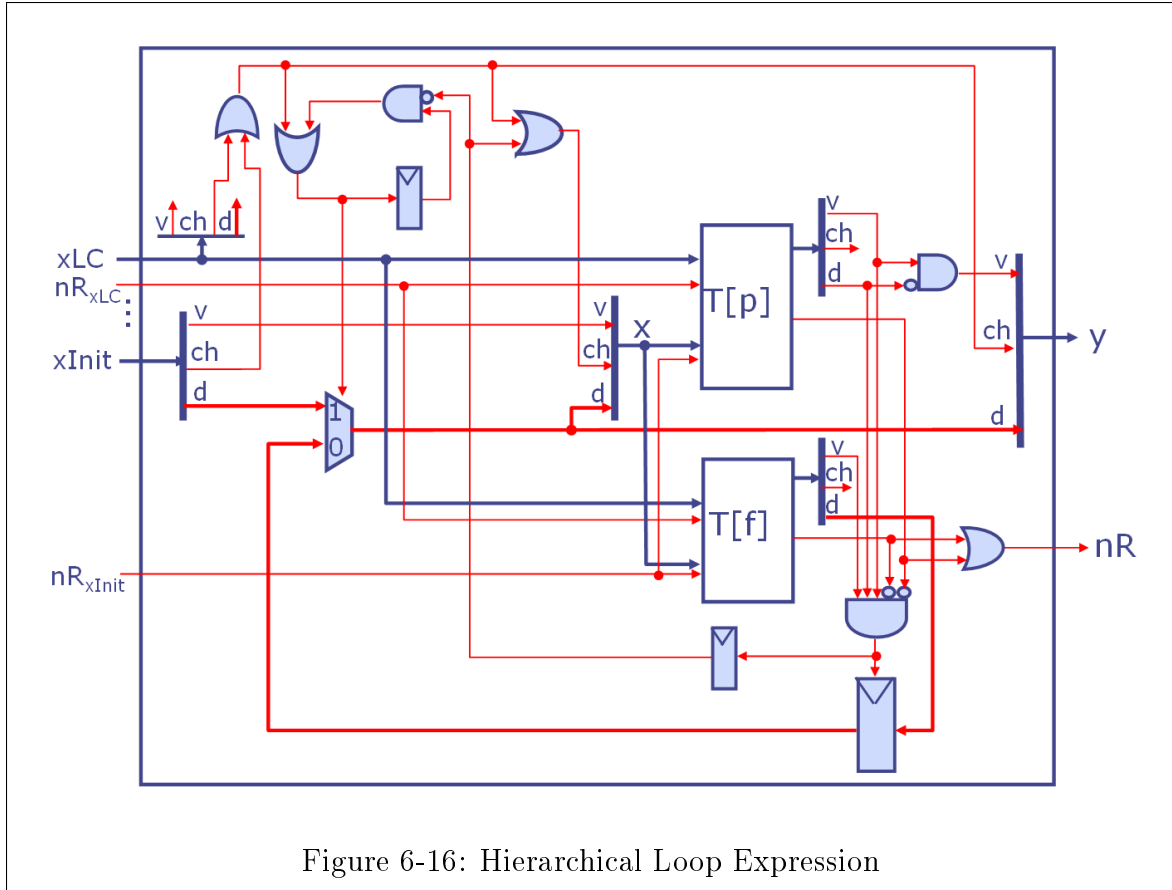
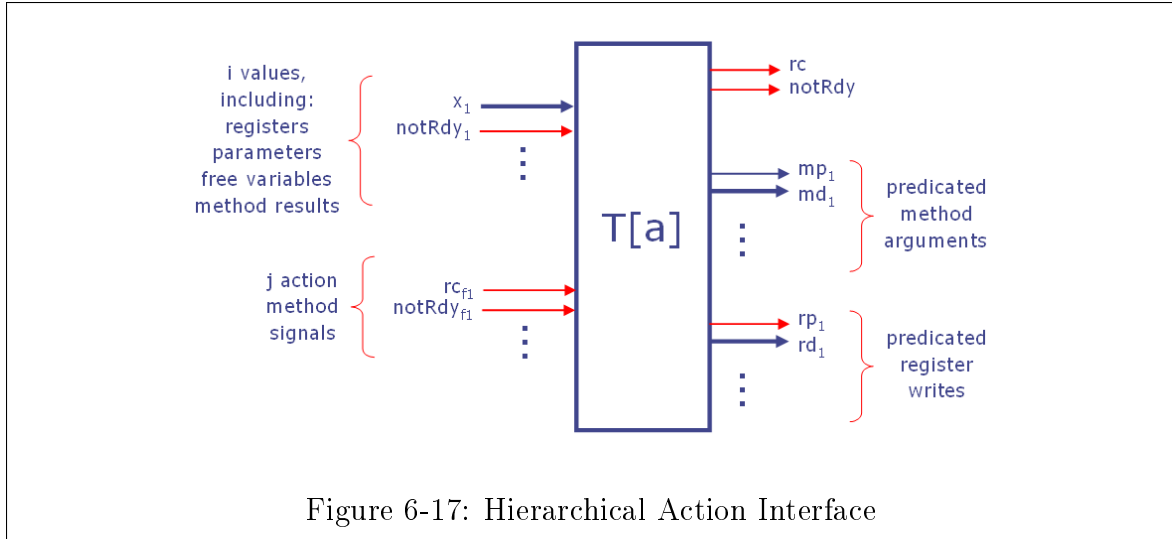


Figure 6-16: Hierarchical Loop Expression

of performing the computation, the loop expression must assert the *changed* wires in the middle of computation. We discussed this in Section 4.4.7. The issue is that if the loop expression contains a call to a value method, it is possible that this method's result will also change in the middle of computation due to a change to the underlying architectural state (before the loop has been reserved), and this change it will be impossible to distinguish between the two changes. If we cannot tell when the underlying values for a method changed due to another action committing, then we cannot rely on these values, and we must wait until the loop has been reserved. Thus in order to be able to trust the results of loops that call methods we would have to add some kind of a loop reset signal to all our interfaces to inform callers when data used for evaluation of value methods has changed. This is different from our current *changed* signal in data 3-tuples which also incorporates the changes in the arguments.



6.7 Hierarchical Actions

6.7.1 Hierarchical Action Interface

Figure 6-17 presents the interface for hierarchical actions. The inputs to actions now include rc and $notRdy$ signals from action methods. Even though these are circuits for actions, they do not perform any actual updates, so they do not need the rsv signal.

The output of actions consists of:

- rc – asserted when the action is ready to commit
- $notRdy$ – asserted when the action evaluates its guard to *false*; when both rc and $notRdy$ are asserted, $notRdy$ takes precedence.
- mp, md – method predicate to indicate that a particular method call is active from the action and method data which carries the argument to the method call
- rp, rd – register predicate to indicate that a particular register should be written when the action commits and the data that should be written

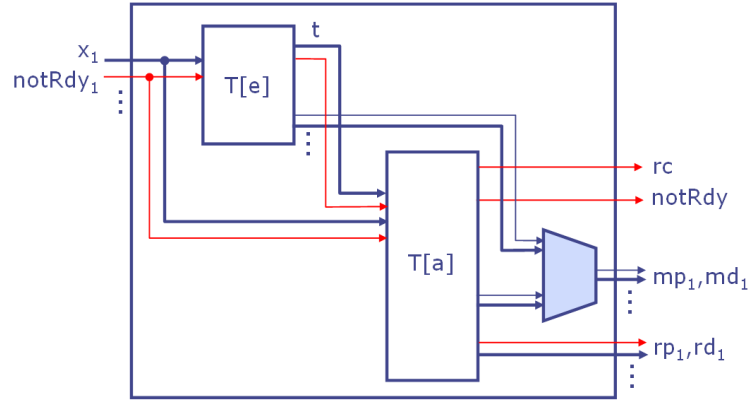


Figure 6-18: Hierarchical Let Action

6.7.2 Hierarchical Let Action

Figure 6-18 presents the circuit for a hierarchical let action. The setup is very similar to that for a hierarchical let expression: the output of the expression is passed into the action, and parameters to value methods called from both blocks are passed through a one-hot MUX.

6.7.3 Hierarchical Register Assignment Action

The hierarchical register assignment action is presented in Figure 6-19. The control and method call wires are simply shuffled in and out of the expression. Register assignment logic is same as in Figure 4-12.

6.7.4 Hierarchical Conditional Action

Figure 6-20 presents our hierarchical conditional action circuit. It is similar to the circuit from Figure 5-13, but also includes method predicate and data wires. We protect the predicate wires from being asserted without the condition evaluating to *true*.

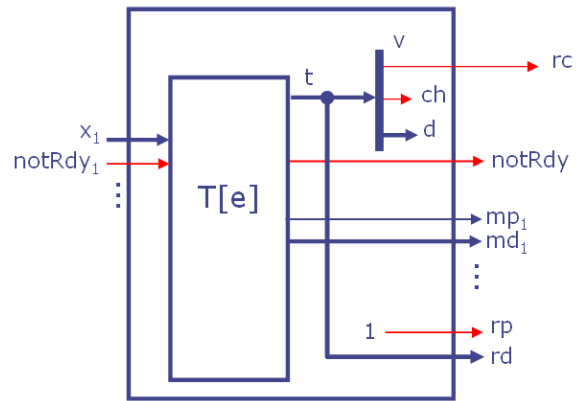


Figure 6-19: Hierarchical Register Assignment Action

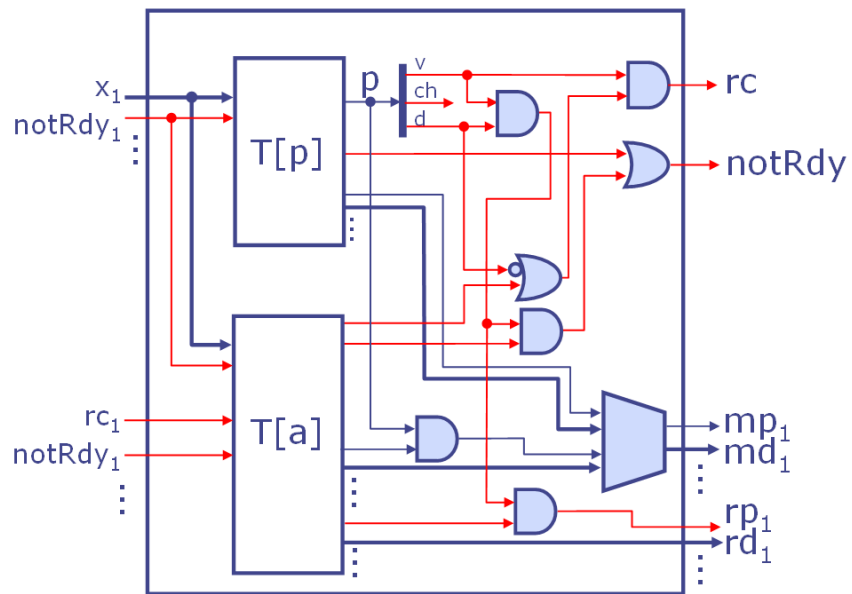


Figure 6-20: Hierarchical Conditional Action

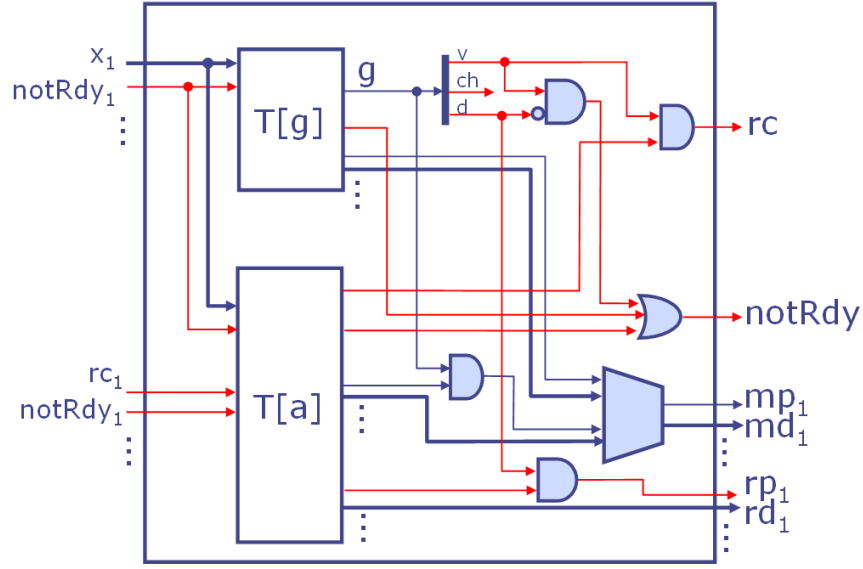


Figure 6-21: Hierarchical Guarded Action

6.7.5 Hierarchical Guarded (When) Action

Figure 6-21 presents our guarded action circuit. It is similar to Figure 5-14, but also includes method predicate and data, which are handled similarly as in Figure 5-13.

6.7.6 Hierarchical Parallel Action Composition

Figure 6-22 presents our parallel composition circuit. We simply combine all the control signals and select the data for method calls and register writes from the two actions.

6.7.7 Action Method Call

Figure 6-23 presents our circuit for action method calls. The method is always called, and we combine the *notRdy* signals from the expression and method call.

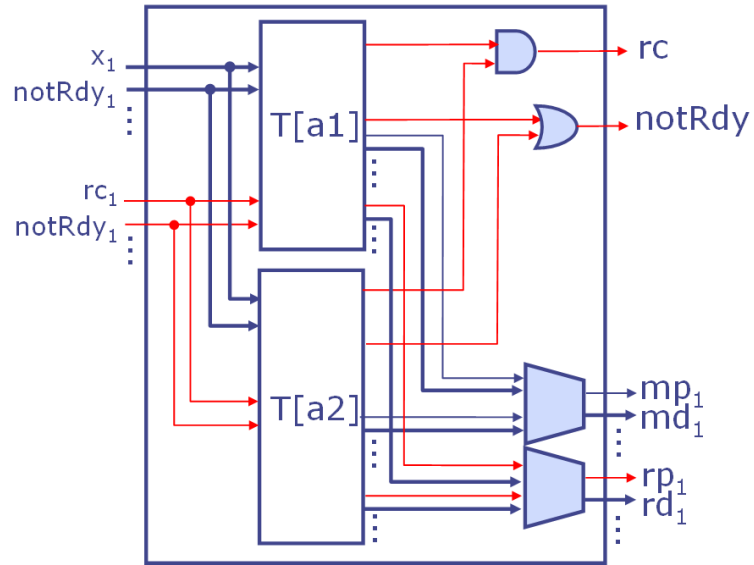


Figure 6-22: Hierarchical Parallel Actions

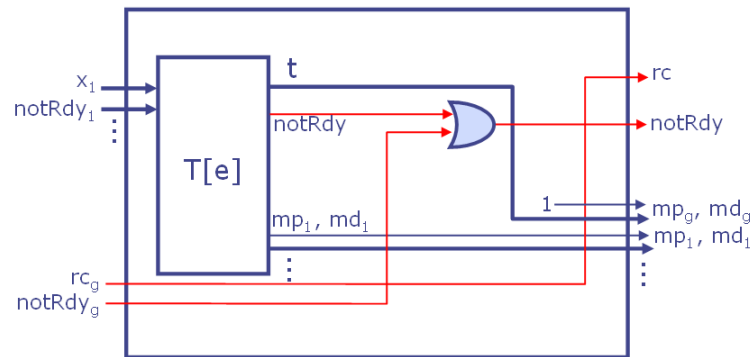


Figure 6-23: Hierarchical Action Method Call

6.8 Module Integration

In previous sections we have described how to build speculative circuits for actions and expressions. We can easily integrate those circuits with our top-level scheduler (simply connect matching wires) and create flat modules. In this section we present circuits for composition of method interfaces from the caller and callee sides.

Figures 6-24 and 6-25 present the circuits for composing callers and callees to value and action methods. For the most part the composition involves connecting matching wires from the method's action or method. In the case when multiple methods or rules can access the same method in a child module, we must combine the *rsv* signals. We use the method predicate wires produced by bodies of rules or methods to select which argument and *cmt* signal to pass on to the method called. All the other signals are simply distributed to the appropriate interfaces. The register-update logic is identical to the flat-module case.

6.9 Summary of Scheduling Restrictions

We end this chapter with a summary of restrictions we must place on modules generated with our technique. These have been discussed before, in this and previous chapters. It is, however, useful to summarize this information in one place, as it provides a single reference for the “rules of the road”. Each restriction is accompanied with a very short explanation of the reason for it.

1. Cannot share calls to action methods in a single clock cycle. Reasons: we cannot compute multiple sequential commits or generate multiple reserve signals.
2. Cannot share calls to methods with parameters in a single clock cycle. Reasons: we cannot pass multiple sets of parameters or compute multiple results.
3. An action can have calls to conflicting methods only if the calls are mutually exclusive. Reason: There is no way to legally complete both calls in a single atomic action.

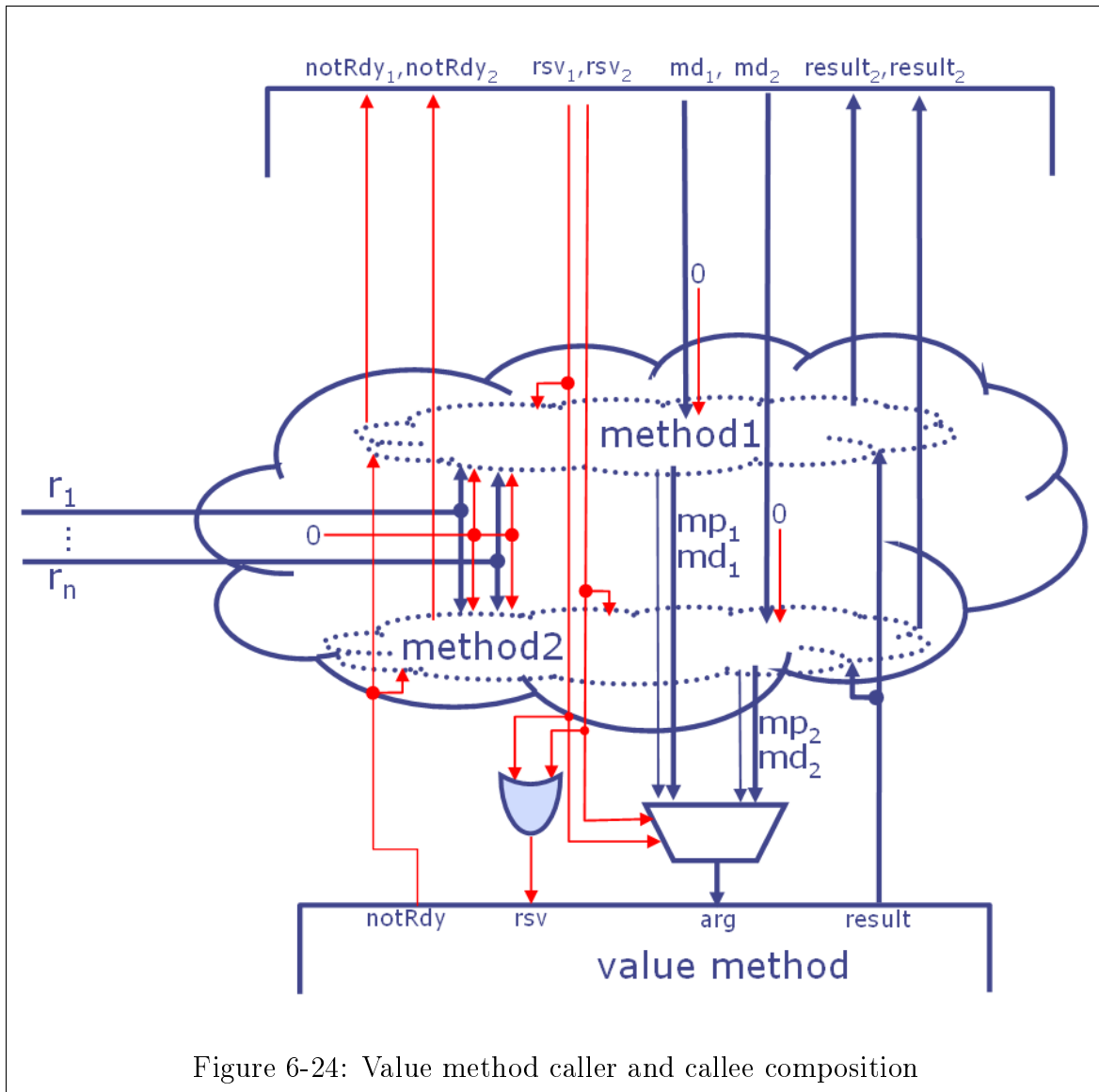


Figure 6-24: Value method caller and callee composition

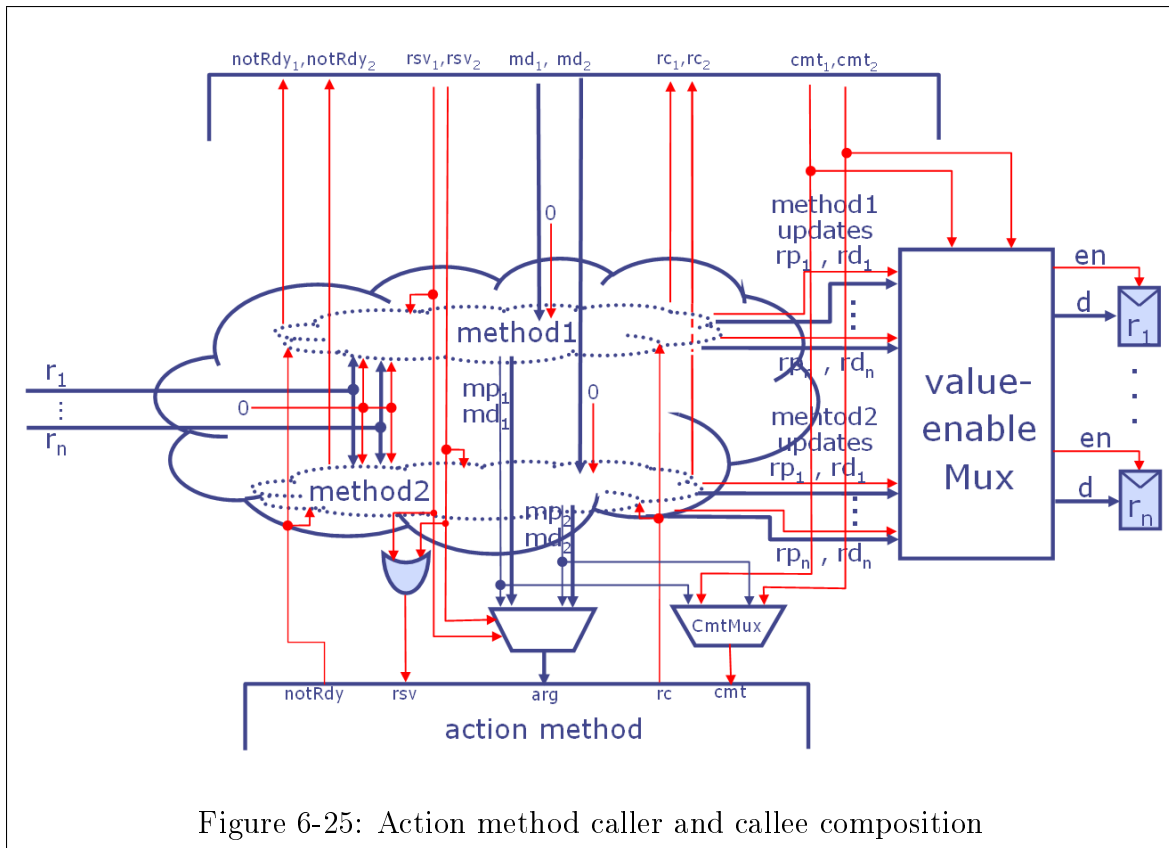


Figure 6-25: Action method caller and callee composition

Chapter 7

Results and Analysis

In this section we discuss some results we have gathered about the performance of circuits generated using our techniques. We have created several designs and compiled them with our compiler in various configurations. We then synthesized them with Synopsys Design Compiler using TSMC 130nm cell libraries.

7.1 Modulo in GCD

Our first example is the GCD circuit originally from Figure 2-2. We have fixed the size of operands to 16 bits. The remainder computation is implemented with a simple divider using 16 compare-subtract-shift stages. We used this basic template to generate different versions of this circuit to allow the remainder to take 1, 2, 4, 8 and 16 clock cycles using `delay` operators. We also factored the remainder operation out into a loop to save area. The different versions of computing the remainder are depicted in Figure 7-1. Our language only allows for construction of `while – do` loops, which test the condition one time more than they compute the body. Thus using the loop construct requires 17 clock cycles to compute the remainder. Adding and using a `do – while` loop construct would remove the spurious clock cycle from our design.

We have elected to use a constant-latency looped divider for the remainder computation, because of its simplicity. We could have just as easily designed one that finished early if possible, but that would require either adding a shifter or storing

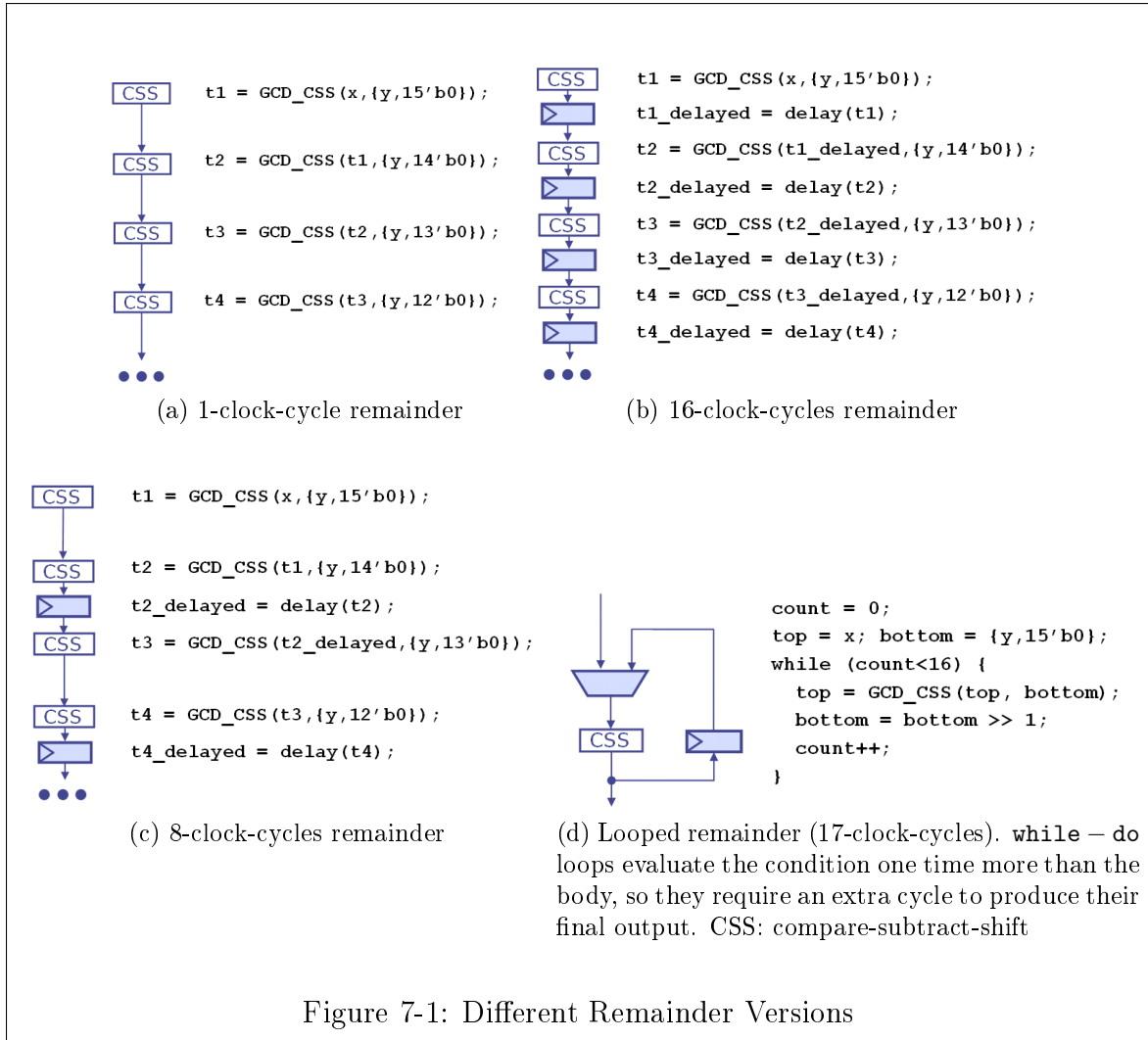


Figure 7-1: Different Remainder Versions

design description	clock cycles per remainder	critical path (<i>ns</i>)	area (μm^2)
GCD - no delays	1	8.92	41,448
GCD - 1 delay	2	5.21	39,807
GCD - 3 delays	4	2.75	44,886
GCD - 7 delays	8	1.63	44,328
GCD - 15 delays	16	0.98	52,372
loop	17 - loop	1.2	10,414

Figure 7-2: Multi-cycle GCD results. Using loop-based remainder significantly reduces the area.

more data on the back edge of the loop, both of which would cost additional area.

Figure 7-2 presents the results of synthesis. Through adding additional **delay** operators to the computation of a remainder, we were able to reduce the clock period from $8.92ns$ to $0.98ns$, a factor of 9.1. This reduction in clock period cost an increase in area from $39,807\mu m^2$ to $52,372\mu m^2$, a factor of 1.3. This area is taken up by the additional registers and overhead logic we inserted for the intermediate data and could be avoided by using a multi-cycle combinational path to represent the **delay** operators. The overall time to compute a remainder has also increased by a factor of $\frac{16*0.98}{8.92} = 1.8$.

The loop version of the remainder has further decreased the area to $10,414\mu m^2$, a total reduction by factor of 5. This is because we no longer need to replicate the compare-subtract-shift logic for 16 stages of the divider. The clock period is $1.2ns$, and the overall time to compute a remainder increased by a factor of $\frac{17*1.2}{8.92} = 2.3$. The significant area savings and relatively fast clock show that introducing a loop can be more desirable than unrolling the computation. That balance may shift towards unrolling the computation with development of pipelining, which is subject of future work, assuming the underlying expression is heavily utilized, or with utilization of multi-cycle combinational paths which would eliminate most of the **delay** registers.

It is worth noting that our approach allows us to easily select the best design-specific trade-off between the clock period, area and throughput.

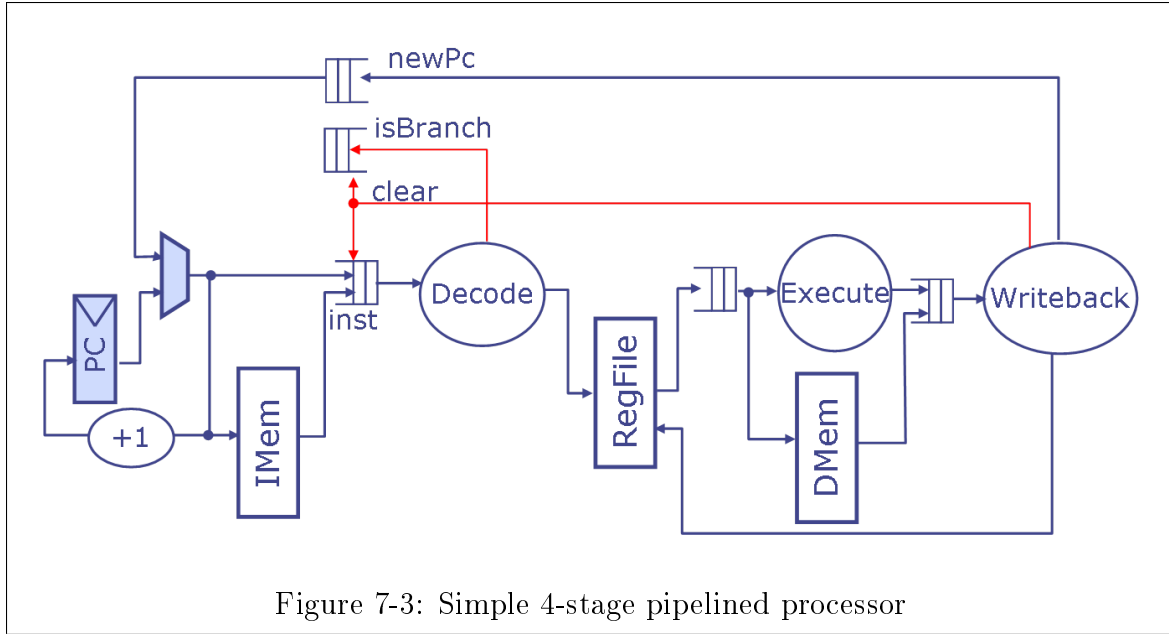


Figure 7-3: Simple 4-stage pipelined processor

7.2 CPU with Division

We have developed a simple 4-stage pipelined RISC processor and compiled it with our compiler. Figure 7-3 shows a high-level pipeline diagram for the processor. We used FIFOs to facilitate communication between pipeline stages. This allowed us to express each stage of the pipeline as a separate rule in BMC, decoupled from all other rules, without having to worry about processing the same data multiple times or overwriting not-yet consumed data between pipeline stages. The IMem and DMem blocks in the diagram are not included in the synthesized circuits, and can take an indeterminate number of clock cycles to respond. Our processor uses 16-bit operands. On seeing a branch instruction, the instruction fetch stage and the decode stage stall till the branch is resolved in the execute stage.

The Execute stage has an expression for executing each instruction. The result from the appropriate expression is selected in the Execute stage using a conditional expression. Various branches of the conditional can take different number of clock cycles – our completion logic correctly determines when the instruction results are ready (see Figure 7-4 for an example).

To illustrate our compiling technique, we initially implemented only simple RISC

instructions: **ADD**, **SUB**, **BRANCH**, **LD** and **ST** – their expressions in the execute stage take only one clock cycle to execute. We then added a **DIV** instruction. The division operation is implemented using a naive shift-compare-subtract divider. There are three versions of this divider. The first one implements a fully unrolled single-cycle divider. The second one implements a fully unrolled divider with 15 delay registers between every operation (the Delayed Divider in the table). This unit performs the division in 16 clock cycles. The third one (the While-loop Divider in the table) implements the compare-subtract-shift operation inside a while loop, thus not duplicating any logic, but adding an implicit delay register for loop data. This divider performs the division in 17 clock cycles, as explained in Section 7.1. No special code had to be added to implement a multi-cycle divider vs single-cycle divider or other single-cycle instructions.

Figure 7-5 shows the synthesis results of our processor. In all the designs, we synthesized for fastest clock period. We used the CPU without a divider as a basis for comparison.

As can be expected, a single-cycle division circuit created a very long combinational path which slowed down the clock by a factor of 6.3. The area of the circuit increased by a factor of 2.1, as the added unrolled logic now constitutes a large portion of our very simple design. By inserting 15 delay registers, one between each step of division, the clock cycle recovered to just 6% slower than the speed of the original design with no divider. But the area increased by another 13%. The divider implemented using a while-loop fixes that flaw as well. The area is now only 24% larger than the original design while the clock is only 4% slower.

We have succeeded in adding a new multi-cycle instruction without increasing the area significantly and without slowing down the clock. We have also demonstrated that using our loop construct results in a significant reduction in area because we no longer duplicate the same circuit just to avoid rule-splitting. Furthermore, our loop implementation is efficient: it does not introduce long critical paths.

It is worth noting that division instructions are often omitted from simple RISC architectures, in part because they would introduce non-uniformity in the implemen-

```

if (instr == ADD) {
    // 1 clock cycle
    result = arg1 + arg2;
} elseif (instr == DIV) {
    // 17 clock cycles
    count = 0; result = 0; top = arg1; bottom = { arg2, 15'b0 };
    while (count < 16) {
        result = result << 1;
        if (top >= bottom) { top -= bottom ; result |= 1; }
        bottom = bottom >> 1; count++;
    }
} elseif ...

```

Figure 7-4: Partial Listing of ALU implementation. The DIV instruction takes 17 clock cycles to complete, while other instructions take only 1 clock cycle. The non-strict if operator selects the correct result and returns it as soon as it is available.

CPU Design	Divide Latency	critical path (<i>ns</i>)	area (μm^2)
No Divider	–	1.04	38,222
Unrolled Divider	1	6.59	81,535
Delayed Divider	16	1.1	92,352
While-loop Divider	17	1.085	47,369

Figure 7-5: CPU Synthesis Results

tation due to their multi-cycle nature. Our methodology allowed us to completely bypass these problems, and allowed us to add the division instruction without significantly modifying any other part of the design.

7.3 FFT

We have built a simple 16-point FFT design. The data is represented in 16-bit fixed point format (8 bits of integer and 8 bits of fraction). We folded the FFT using nested loops (one for each level of butterflies and one for each row) to generate only a single butterfly circuit inside a nested loop. See Figure 7-6 for details of the design. A similar folding in Bluespec is also described in [19]. We then manually unrolled the inner loop by a factor of 2, 4 and 8. The 8-butterfly design results in having an

entire level of FFT done in a single clock cycle and eliminates the inner loop. We then unrolled the outer loop 2 and 4 times, resulting in 16- and 32-butterfly designs. The last version resulted in a purely combinational circuit. We then inserted 1 and 3 **delay** registers between butterfly levels of the combinational version to compare it to the looped versions. Finally, we collapsed the nested loops of the original 1-butterfly design into a single loop which keeps track of both the current level of FFT being computed and the butterfly in that level. Results are in Figure 7-7.

The property of **while – do** loops requiring an extra clock cycle is particularly inefficient in nested loops, where the inner loop requires an extra cycle for every iteration of the outer loop. The nested-loop 1-butterfly design requires 37 clock cycles to complete its computation, even though the butterfly logic is being utilized in only 32 of those cycles. This means that during 13% of clock cycles the butterfly logic sits idle, a significant drain on throughput. Collapsing the nested loops into a single-nest loop results in latency of 33 clock cycles, which means that the butterfly logic sits idle only 3% of clock cycles. We also get a reduction in area size of 25%, because only need one set of registers on the back edge of the loop. This transformation is similar in its goal to loop merging in parallelizing compilers.

For the most part, the results are exactly what we expected them to be. Using a loop to express a computation allows us to significantly reduce the area, compared to a fully-unrolled computation. Building designs using a loop is extremely easy – no more difficult than a building a fully-unrolled version.

We have explored trading area for throughput by adding more butterflies in a single row in the inner loop. As expected this expanded the size of the circuit, but had a small effect on the clock period. This is because the butterfly computation is all done in parallel. We were able to vary the critical path from $1.96ns$ to $8.26ns$ and the area from $68,673\mu m^2$ to $335,730\mu m^2$ by unrolling the outer loop.

The designer can select the most appropriate implementation from the variety of offerings, depending on design parameters.

Adding enough logic to complete a full level of butterflies in a single clock cycle removes a loop level, which removes loop registers and MUXes. The additional com-

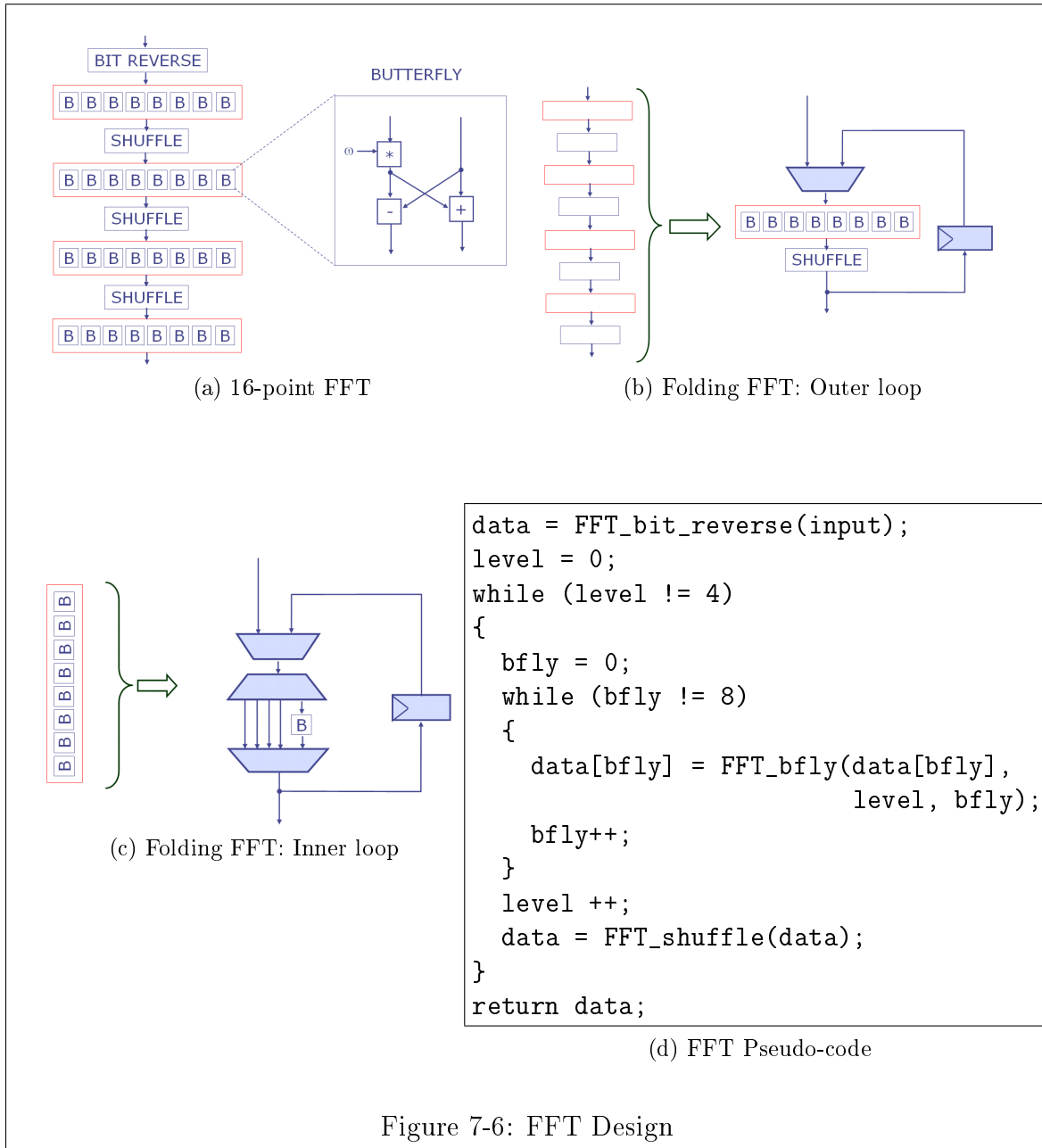


Figure 7-6: FFT Design

FFT Design	FFT Latency	critical path (ns)	area (μm^2)	throughput (FFT/s)	throughput (FFT/s/ μm^2)
1 butterfly	37	2.70	91,622	10,010,010	109.3
2 butterflies*	21	2.75	124,032	17,316,017	139.6
4 butterflies	13	2.91	196,331	26,434,047	134.6
8 butterflies*	5	2.59	190,156	77,220,077	406.1
16 butterflies	3	4.40	335,730	75,757,575	225.7
32 butterflies*	1	8.26	150,325	121,065,375	805.4
32 butterflies*	2	3.53 (1.17)**	217,217	141,643,060	652.1
32 butterflies*	4	1.96	237,776	127,551,020	536.4
1 butterfly single loop*	33	2.67	68,673	11,349,450	165.2

Figure 7-7: FFT Synthesis Results. *Pareto points. **3.53ns was obtained by synthesizing with a test harness. 1.17ns is anomalous and was obtained by synthesizing without a test harness. Likely cause of the anomaly is that without a testing harness the circuit has exactly one register and no register-to-register path.

putational logic is offset by removal of loop logic, but the critical path is reduced. The 8-butterfly version is preferred to the 4-butterfly version in every way.

Unrolling the outer loop and adding full levels of butterflies results in longer critical paths. The reduction in area when going from 16-butterflies to 32-butterflies is surprising at a first glance. Closer inspection shows that full elaboration was able to remove more than half of the multipliers in the butterflies because of multiplying by 1 or i . Furthermore, full expansion of the algorithm resulted in elimination of all loops which reduced amount of MUXes and registers. A similar observation has been made in [19]. Despite all these optimizations, the critical path remains quite long.

When synthesizing the 2-clock-cycle 32-butterfly design, we have obtained an anomalous critical path result of 1.17ns. We re-synthesized the design together with the testing harness and the critical path increased to a more expected 3.53ns. The problem appears to have been that the synthesis tool was confused about the critical path with no register-to-register path. The area provided ($217,217\mu m^2$) is from the original synthesis without a testing harness.

Our compilation scheme does suffer from a potential problem. We would like to use a register file for storing intermediate results in the loop, but our synthesis method only allows us to use flat registers on the back-edge of loops. In general we cannot complete actions inside a loop, or we risk losing atomicity. Thus generating the loops necessary for computation of the FFT required that we update the state of the entire FFT in each loop iteration. This results in very large loop registers and MUXes which waste area and power. The desired solution of using a register file for storing intermediate results would address both these problems, but requires a significant additional compiler analysis. In some cases, it may even be possible to reuse existing architectural register files.

This FFT design also shows that the while-do loops of our language can waste significant number of clock cycles. Converting the nested loops into single loops would eliminate 4 clock cycles from the 1-butterfly version. The real solution to the problem of wasted cycles, though would be to introduce `do-while` loops to the language. This should be a fairly straight-forward exercise, given the work in this thesis.

7.4 Viterbi

Viterbi algorithm is well studied and known to be an interesting case for circuit design[43, 24]. The standard design for Viterbi is to split the computation into two steps: the path metric, or forward, computation step and the trace-back step. We have selected a design with a 4-bit encoder (8 states, $k = 3$), and a 20-step trace-back path (using rule of thumb of $n = 5 * (k + 1) = 20$).

The key to building an efficient Viterbi design is to match the rates at which the path metric unit (PMU) consumes inputs and the trace-back unit (TBU) produces outputs. Our exploration attempts to match these rates as closely as possible. We have the following levers at our disposal: adding add-compare-select (ACS) units to the PMU, increasing the number of bits the TBU is able to process in a single clock cycle, and increasing the number of trace-back steps performed beyond the minimum

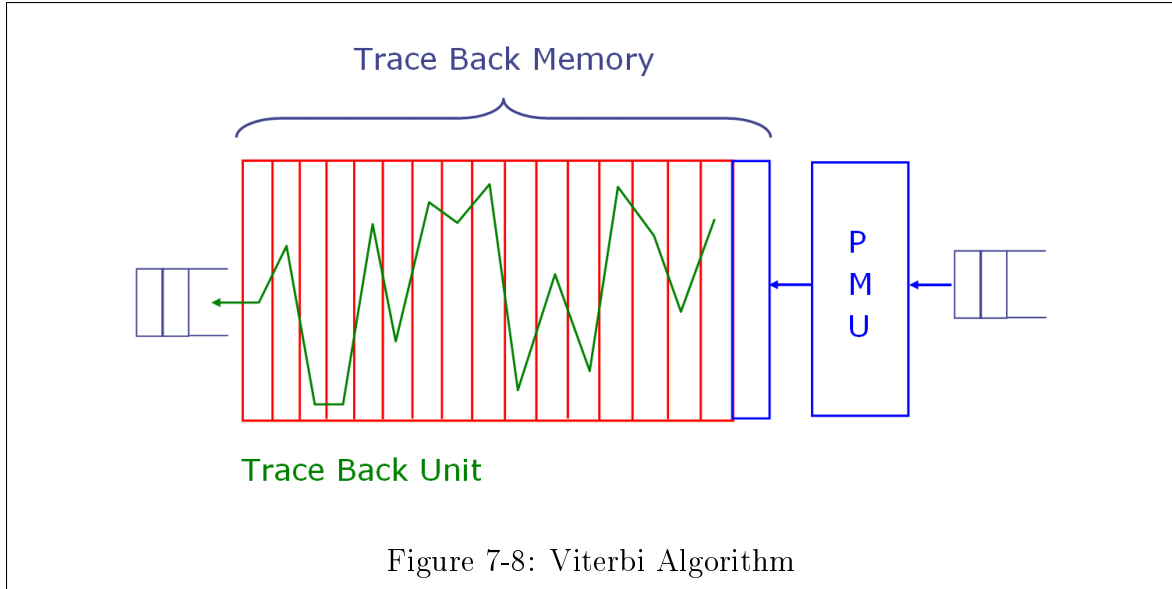
of 20 (thus increasing the number of bits produced by the TBU for each full trace).

We store the trace-back data in a register file. Each invocation of the path metric computation writes a single entry in the register file, corresponding to a number of radices computed. Performing each full trace-back reads every entry in the register file. Technically this is not allowed in the work presented in this thesis, because the register file is represented as a module. The reason we do not allow calls to methods in child modules from loops is that we have not presented a circuit for resetting loops when data in the child module is updated, as has been described in Section 6.6.9. Such an enhancement is fairly simple, but not even necessary in this case: we are able to trivially prove that each time the register file is updated by the PMU, the loop in the TBU resets automatically. The PMU writes the index of the register with youngest data and the loop in the TBU consumes this index. Adding necessary analysis and loop-reset logic to permit use of child modules in loops is subject of future work.

Our designs exploit an interesting feature of circuits generated by our design methodology: we can perform computation for methods and rules which are not yet ready to fire. This means that we can overlap the execution of the PMU and the TBU, even though the PMU cannot update the trace-back memory until the TBU has completed its work. This means that if the TBU is faster than the PMU, its computation will be completely hidden by the PMU. If the PMU is faster than the TBU, TBU's computation will be hidden (with an additional clock cycle required for synchronization).

Figure 7-8 presents the top-level flow of the Viterbi Algorithm. Figure 7-9 presents our results.

We have explored a total of 6 different designs. Each of our designs decreased the average number of clock cycles needed to produce a bit of output while operating in steady state. To produce each successive iteration we selected the part of the design that was a bottleneck (PMU or TBU) and improved its throughput. All designs, but the second last one match one iteration of the forward step with one iteration of the trace-back algorithms. The 8-ACS, 2-radix design needs to perform



the forward step twice in order to supply enough data for the trace-back algorithm. While this approach slightly decreases the average number of clock cycles to produce a bit of output, it also increases the clock cycle by enough to offset the per-clock-cycle throughput gains.

Overall, we were able to vary the critical path from $1.61ns$ to $3.13ns$, while increasing the throughput from $28,232,637bps$ to $91,282,519bps$. Once the design has been explored, it is up to the designer to select the most appropriate version.

All the designs of Viterbi were written with our `while – do` loops and would increase throughput by using `do – while` loops. The PMU uses nested loops and we could reduce area and improve critical path by reducing the nested loops into a single-nest loop.

It is worth noting that the design time for Viterbi from the first line of code to the last result collected was 4 days, including fixing a subtle bug in the compiler.

7.5 Analysis

Our approach to design exploration and finding best-fitting tradeoff for the needs of the overall design seems to be effective. We are able to rapidly modify the design of various computational components to shorten the critical path, reduce area or

# of ACS	# of FS radixes	cycles for FS	TB steps per cycle	output bits per TB	cycles per TB	cycles per output bit	critical path (ns)	area (μm^2)	throughput (bps)
1*	1	21	1	1	21	22.0	1.61	22,088	28,232,637
2	2	27	1	2	22	13.5	2.05	28,905	36,133,695
4*	2	11	2	2	12	6.5	2.52	27,806	61,050,061
4*	2	11	4	2	7	5.5	2.45	31,882	74,211,503
8	2	7	2	4	13	5.25	3.14	34,907	60,661,207
8*	4	13	2	4	13	3.5	3.13	37,108	91,282,519

Figure 7-9: Viterbi Synthesis Results. FS means forward step. TB means trace-back. *Pareto points.

improve throughput. All that exploration was done without changing the semantics of the operation or giving a second thought to how the particular component would be used in the greater design.

While analyzing the results, we have identified two potential concerns with our synthesis approach:

- area overhead due to added flip-flops
- possibility of long combinational paths in control logic

We analyze these two concerns below.

7.5.1 Area Overhead due to Added Flip-Flops

Our synthesis scheme required addition of some storage bits which are used to maintain the state of the computation. Most of these storage bits are related to keeping track of validity of data propagated through the design. A much smaller amount of storage is used to keep track of status of computation in loops. We do not consider registers inserted to keep the actual data being computed to be overhead: those registers are being inserted at a specific request of the designer. Our tool generates registers for data in `delay` operators, but a smarter tool can also generate multi-

cycle combinational paths. The registers generated on the back-edge of loops are unavoidable, so they are also not considered overhead.

The overhead registers fall into three categories:

- flip-flops storing the write-enable signal to architectural registers
- flip-flops storing the *valid* signal in `delay` registers
- flip-flops tracking *firstIter* and *nextIter* state in loops

The first two categories of overhead flip-flops are related. First, it is important to note that each requires only one bit of storage, no matter how wide the data is. This means that the overhead is at most 50% (when data is also one bit) but would be just 3% when the data is 32 bits. Second, for each of these categories, some of the overhead bits can be optimized out. In our GCD example, both x and y are always written at the same time. This means that we only need one bit to keep track of the write-enable signal for both of them. In practice this can lead to a significant reduction in overhead. Synthesis tools can perform this optimization without changing our synthesis methodology.

The last category turns out to be a fairly small contributor. Practical designs have a limited number of loops, and each loop has just two bits of extra storage introduced as overhead.

Overall, we have found that storage overhead was not a big issue in our designs. The worst-case overhead for our CPU had 23 bits of overhead out of 1173 bits total (2.0%) in the 16 clock cycle division version. The worst-case overhead for the FFT design was 4 bits of overhead out of 1031 total (0.4%) in the 4-clock-cycle version. And the worst-case overhead for the Viterbi algorithm was 8 bits of overhead storage out of 422 bits total (1.9%) in the 1 ACS version.

7.5.2 Long Combinational Paths in Control Logic

Our design methodology and synthesis approach has focused on controlling the critical path for the fundamental computation being performed. This is the logic that

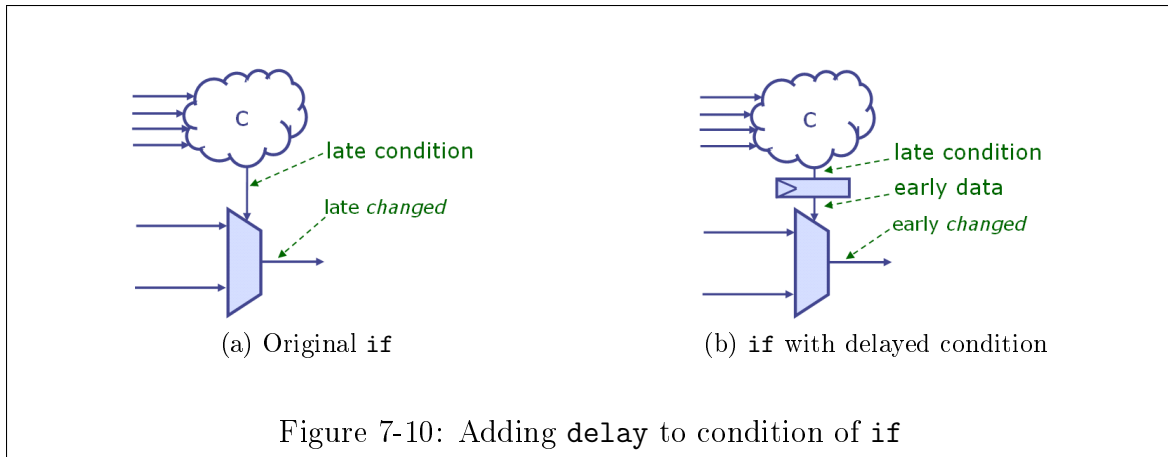
is fundamental to the design, no matter how it is compiled. But we have not done any explicit work to reduce the critical paths in control logic. Under extreme circumstances, it is possible that the control logic will come to dominate the critical path of synthesized circuits. It is important to note that we have not observed this problem (critical paths in our circuits have not included control signals).

From prior work we know that rule scheduling logic is efficient [23]. We use the same approach to creating our Reservation Scheduler and Commit Scheduler, so we expect that logic to also be efficient. *valid* signals are passed through flip-flops in `delay` operators, so their critical path is controllable.

It is possible, however, for the generation of the *changed* signal to create a long combinational path. This is because the *changed* signals are expected to propagate completely throughout the entire circuit in a single clock cycle. Most operators do very little to the *changed* signals: they simply combine the *changed* signals from all the inputs and output a disjunction of them. This has the potential to create long critical paths, but those paths can be optimized using associativity and idempotency of disjunction. An expression computing an answer based on n separate inputs only needs $\log(n)$ gate delays to combine all the *changed* signals. Synthesis tools can perform this optimization without changing our synthesis methodology.

The non-strict `if` operator, however, presents a more serious challenge. In our context, a non-strict `if` does not assert output *changed* signal if the unselected input's *changed* signal is asserted. This means that the non-strict `if` must use the condition in computation of the output *changed* signal, which leads the output *changed* signal to potentially arrive late in the cycle.

One solution to this problem would be to convert the non-strict `if` into a strict `if`. A strict `if` would assert the output *changed* signal even if only the non-selected input asserted its *changed* signal. Thus the output *changed* signal of a strict `if` is not dependent on condition value, and should arrive earlier in the cycle. Implementing a strict `if` is easy: a strict `if` is simply a 3-input primitive operator. While simple, this approach may not always be appropriate: strict `if` operators have useful properties that the design depends on: it could essentially revert the circuit to always produce



results with worst-case delays, just like in Chapter 3.

Another approach to solving such a problem can be to wrap the conditional express in a **delay** operator. That is convert `if c then e1 else e2` into `if delay(c) then e1 else e2`, as depicted in Figure 7-10. This solution may cause a delay of the output by an additional clock cycle, but would still take advantage of the non-strict `if` operator.

If neither of the approaches above is acceptable for the design, then the design may need to be changed in a more substantial way. The potential for long combinational paths on the *changed* signals can be a real problem, even if we have not seen it in our experiments. A better approach to tracking validity of partially computed results would be desirable.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this thesis we have extended the guarded atomic action design methodology with new constructs to enable easy creation of designs utilizing rules which require many clock cycles to complete. We have introduced the multi-cycle `delay` operator and dynamically controlled loops. Unlike previous attempts at multi-cycle design methodology, these new constructs introduce user invisible, micro-architectural state, rather than utilize the user-visible architectural state. This distinction was key to our success, as it eliminated the need to keeping track of shadow state, and enabled concurrent computation of updates for multiple rules.

The introduction of dynamically-counted loops makes it possible to express designs in a more succinct fashion that takes less area than previous attempts. We have demonstrated this by factoring out the logic of several designs such as remainder computation and FFT. While our examples may be quite simple, they demonstrate an ability to solve a common and tedious problem: paths which are inherently slow but not worth optimizing aggressively because they are not common.

We have combined these enhancements with non-strict `if` operator. Using non-strict `if` enables the designer to easily exploit the power of unbalanced paths on different branches of the `if`, demonstrating the power of multi-cycle design. We utilized the non-strict `if` to add a multi-cycle instruction to a CPU without having

to introduce any new control or synchronization logic.

We were careful to ensure that circuits generated with our technique compute all updates optimistically. That is a rule will continuously compute its updates without necessarily being reserved. Those updates can be utilized when the rule is reserved to improve performance. This is an extension of the Hoe-Arvind approach, where all rules always compute their updates irregardless of which ones are enabled by the scheduler. In order to accommodate multi-cycle rules, our approach is more complicated than Hoe-Arvind.

Finally, we have presented an efficient synthesis technique which generates circuits from designs utilizing our new construct as well as the basic guarded atomic action constructs. Our technique is able to handle modular, hierarchical designs, which enables design with basic libraries, such as FIFOs, register files, etc.

We have demonstrated that our extensions and synthesis approach allow for optimization of the critical path without changing the semantics of the underlying design. We were able to explore the latency and throughput of different implementations of algorithms in a design, all without changing the interfaces for those algorithms. Furthermore, our language extensions significantly improved the expressiveness of the language, enabling rapid design exploration, even for algorithms widely considered to be complicated and difficult to optimize, such as the Viterbi algorithm. Because of the close correlation between the design language and the hardware generated by our compiler, design exploration gives us predictable results after synthesis – a feature critical to enable designers to utilize their creativity in a productive manner.

We have identified several avenues of improvement, which are presented next. Overall, it appears clear to us that the greater guarded atomic action community, and Bluespec community in particular, would benefit from multi-cycle language extensions and synthesis.

8.2 Future Work

Throughout the work on this thesis, many ideas for improving the current methodology were identified. Some of the ideas have already been discussed above, including flattening of nested loops, introducing `do – while` loops, allowing method calls from inside loops and automatic generation of multi-cycle combinational paths out of expressions with `delay` operators. Below is a discussion of other ideas for enhancements and new directions for the work presented in this thesis.

8.2.1 Pipelining

One of the common methods of increasing throughput of a circuit is to pipeline it. The principle of pipelining is to allow computation on multiple data to be performed at the same time, each at a different stage of a circuit. One of the methods we use for critical path reduction – inserting `delay` operators – can produce circuits that have as much area as fully pipelined logic, but lack the control logic to perform computation on multiple data. The remainder computation in our GCD example has this property, as does the division instruction in the CPU. It would be quite desirable to enable the ability to fully pipeline logic with long critical paths and exploit the ability to process multiple data concurrently. There exist several challenges to enabling such an ability. We will illustrate the general problem by using the GCD computation as an example.

In the GCD example, each remainder computation (invocation of *GCD* rule) requires the full result of the previous remainder computation to be present. Thus, in order to process multiple data in the remainder computation, we would need to enable the entire module to accept multiple input data, keep track of which inputs have been fully processed and are ready to output, and output them in the first-in first-out order. One way to do this would be to introduce an operator which pipelines an expression automatically, and force the designer to implement the logic of keeping track of multiple data. A more desirable way to do this would be to allow the designer to annotate the *doGCD* module with information on the flow of data in it, and enable the compiler to perform analysis and produce a circuit which

automatically keeps track of all the data being processed concurrently.

8.2.2 Modular Scheduling

The current Bluespec solution to hierarchical synthesis is to enable every module to contain a separate scheduler for the rules in the module[23, 34]. The schedulers work together to make sure that evaluation of all rules maintains atomic consistency. The solution presented in this thesis applies a rule lifting procedure, puts all the rules in a single top-level module and uses a single scheduler to schedule all rules. Both approaches have their advantages. In particular, the Bluespec approach can fail to schedule some combinations of rules and method call trees as efficiently as the approach presented in this thesis. On the other hand, the approach presented in this thesis can generate more scheduling logic and increase the time to producing a schedule. It would be desirable to port the Bluespec approach to our synthesis approach to allow the designer a selection of choices.

8.2.3 Multi-cycle Control Signals

We have discussed previously the potential problem with generating long combinational paths for *changed* signals. We would like to develop a solution to this problem, presumably by allowing the *changed* signals, or some version of them, to propagate over multiple clock cycles.

8.2.4 Combine with Ephemeral History Registers

Ephemeral History Registers [6] (EHRs) are the conceptual opposite of our work. The idea of EHRs is to allow multiple rules to perform their computation in a single clock cycle, in an order that forces introduction of bypassing of data written by the earlier rule to the later rule. That is if rules A and B have a relationship of $A < B$, we want to compute B then A in a single clock cycle.

The result of using EHRs is that the critical path of the circuit can increase due to chaining of computation. This is not always the result, though. Sometimes the

conflicting computation being performed is quite simple (like writing a constant to a register) so the only overhead could be the additional cost of a MUX. Furthermore, this MUX may not even be on a critical path. In such a situation, it could be quite beneficial to allow conflicting rules to bypass data to each other, because it could save a clock cycle from the computation.

We encountered such a situation in our Viterbi design. The rules for PMU and TBU are conflicting because they both manage the circular buffer formed by the Trace Back Memory. The source of conflict is reading and writing of pointers to this circular buffer. If we were able to bypass the data written by PMU to be read by the TBU, we would not need a clock cycle delay between completion of TBU rule and completion of PMU rule. The crux of the computation of these two rules would still proceed as it currently does. Only at the time that they are both ready to commit, would we see the difference: they would both commit in a single clock cycle, rather than the current one rule at a time.

8.2.5 Synthesis of Asynchronous Logic

The methodology used in this thesis exploits the fact that the semantics of atomic actions are untimed, that is, the time to execute an action does not change its outcome. Due to this property, we were able to introduce operators which may have increased the number of clock cycles necessary to compute an expression, but did not change the final result in the reference one rule at a time execution model. The specific implementation of the `delay` operator presented here is to introduce a register to store an intermediate value. One suggested future work improvement is to instead convert expressions using `delay` operators into multi-cycle combinational paths. It is, however, possible to take this suggestion much further. We could completely remove the dependency on the clock and instead synthesize asynchronous circuits. The common method of implementing asynchronous computation is to treat data as tokens, which are carefully passed around the circuit with request-acknowledge logic [20]. From the high-level point of view, this is quite similar to data flow.

Asynchronous circuits offer many potential advantages over their synchronous

counterparts:

- power can be reduced due to elimination of the clock tree and all related PLLs
- power can be further reduced by only performing the necessary computations. A similar effect can be achieved with synchronous circuits by employing more advanced techniques, like clock gating.
- many computations can be implemented more efficiently with asynchronous circuits because they must compute their completion signals explicitly rather than rely on an externally imposed clock. For instance an asynchronous ripple-carry adder completes its computation in average $O(\log(n))$ time, while a similar adder in a synchronous circuit must assume completion in worst-case scenario of completion in $O(n)$ time.
- computation can be performed faster because of elimination of safety margins imposed on propagation of clock signals due to clock skew and jitter, necessary to ensure that chips function correctly under a variety of process and environmental variations
- computation can be further sped up because the results of a computation can be propagated as soon as they are available – there is no need to wait for the next clock cycle to capture a result in a register

Unfortunately, there are several barriers to implementing atomic action systems efficiently in a data-flow-like manner.

In order to achieve many of the promised savings of asynchronous circuits, the circuit must be carefully designed to leverage the inherent advantages of asynchronous design. Perhaps the biggest difficulty is in reducing the amount of wasted computation being performed and in resetting the circuit for computing an expression when it is deemed to be wasteful, all while exploiting as much parallelism from the design as possible.

Consider a design comprising of two conflicting rules $R1$ and $R2$. The guard for $R1$ is $g1$ and the guard for $R2$ is $g2$. Suppose that both rules are inactive at a particular

time. The scheduler needs to evaluate $g1$ and $g2$ and select a rule to activate. If it attempts to evaluate both $g1$ and $g2$ at the same time, and $g1$ completes first and evaluates to *true*, presumably the scheduler should activate $R1$. At that point we still have $g2$ trying to complete its computation. We could try to reset the circuit for $g2$, but that would require ensuring that all tokens active in $g2$ are effectively killed – a potentially difficult and expensive operation in asynchronous circuits. We could allow $g2$ to complete, but we do not know how long that may take. In our thesis, we have chosen to allow $g2$ to continue its computation while also using *valid* and *changed* signals to effectively reset computation of $g2$ when its inputs change. Asynchronous design precludes a light-weight implementation of such a solution.

If we allow each guard to complete its evaluation and schedule the rules independently, we must somehow handle a case where they both complete at the same time. This requires an arbiter circuit, which is generally to be avoided. If the design has more rules, we also have to consider if we want to activate one rule at a time – which would slow down the activation of rules – or if we want to wait for multiple rules to evaluate their guards – which would require even more complex arbitration logic. Waiting for all rules to complete evaluation of their guards would avoid that logic, but again would force us to wait until all guards have evaluated before scheduling – which may be too slow.

Another problem with our example is the consideration of when to start evaluating the bodies of $R1$ and $R2$. If we start evaluating the bodies as soon as we start evaluating the guards (which would complete the actions faster) we must again figure out a way to kill the evaluation in progress for the rule which will not fire. If we wait with computation of the body until the guard has been evaluated we may lose too much performance.

Some of these problems can be overcome by utilizing some knowledge about the inherent properties of the computations we are performing. Perhaps we could analyze $g1$ and $g2$ and determine that $g1$ will evaluate significantly faster. We can then implement a scheduler which expects $g1$ to complete and potentially activate $R1$. If we cannot activate $R1$, we can then wait for $g2$ to complete and consider $R2$ for

activation. Only $g2$ would need a reset mechanism. If, on the other hand, $g1$ and $g2$ are estimated to have similar delays, we could implement a scheduler which waits for both to complete before scheduling. Neither $g1$ nor $g2$ would need to be reset. Such an approach can alleviate some of the difficulty with generating efficient asynchronous circuits, but does not address all the problems described above.

Bibliography

- [1] C-to-Silicon Compiler. <http://www.cadence.com/>.
- [2] Handel-C. <http://www.mentor.com/products/fpga/handel-c/>.
- [3] SpecC. <http://www.cecs.uci.edu/specc/>.
- [4] Symphony. <http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/SymphonyC-Compiler.aspx>.
- [5] SystemVerilog. <http://www.systemverilog.org>.
- [6] *2nd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2004), 23-25 June 2004, San Diego, California, USA, Proceedings*. IEEE, 2004.
- [7] *2nd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2004), 23-25 June 2004, San Diego, California, USA, Proceedings*. IEEE, 2004.
- [8] AccelDSP. <http://www.xilinx.com/tools/acceldsp.htm>.
- [9] Jonathan Babb, Martin Rinard, Csaba Andras Moritz, Walter Lee, Matthew Frank, Rajeev Barua, and Saman Amarasinghe. Parallelizing applications into silicon, 1999.
- [10] Prithviraj Banerjee, Malay Haldar, Anshuman Nayak, Victor Kim, Vikram Saxena, Steven Parkes, Debabrata Bagchi, Satrajit Pal, Nikhil Tripathi, David Zaret-sky, R. Anderson, and J. R. Uribe. Overview of a compiler for synthesizing mat-

- lab programs onto fpgas. *IEEE Transactions on Very Large Scale Integration Systems*, 12:312–324, 2004.
- [11] Gérard Berry. *Esterel on hardware*, pages 87–104. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.
 - [12] Bluespec System Verilog. <http://www.bluespec.com>.
 - [13] F. Boussinot and R. de Simone. The esterel language. *Proceedings of The IEEE*, 79:1293–1304, 1991.
 - [14] Mihai Budiu and Seth Copen Goldstein. Compiling application-specific hardware. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 853–863, Montpellier (La Grande-Motte), France, September 2–4 2002.
 - [15] Timothy Callahan and John Wawrzynek. Adapting software pipelining for reconfigurable computing. In *In Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 57–64. ACM, 2000.
 - [16] Catapult-C. <http://www.mentor.com/esl/catapult/>.
 - [17] Nirav Dave. Designing a reorder buffer in bluespec. In *MEMOCODE*, pages 93–102, 2004.
 - [18] Nirav Dave, Arvind, and Michael Pellauer. Scheduling as Rule Composition. In *MEMOCODE '07: Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, pages 51–60, Washington, DC, USA, 2007. IEEE Computer Society.
 - [19] Nirav Dave, Michael Pellauer, S. Gerding, and Arvind. 802.11a transmitter: a case study in microarchitectural exploration. In *MEMOCODE*, pages 59–68, 2006.

- [20] A. Davis and S.M. Nowick. An Introduction to Asynchronous Circuit Design. Technical Report UUCS-97-013, Computer Science Department, University of Utah, September 1997.
- [21] Giovanni De Micheli, David Ku, Frederic Mailhot, and Thomas Truong. The olympus synthesis system. *IEEE Des. Test*, 7:37–53, September 1990.
- [22] Stephen A. Edwards. High-level synthesis from the synchronous language esterel. In *IWLS*, pages 401–406, 2002.
- [23] Thomas Esposito, Mieszko Lis, Ravi Nanavati, Joseph Stoy, and Jacob Schwartz. System and method for scheduling TRS rules. United States Patent US 133051-0001, February 2005.
- [24] G. Feygin and P. Gulak. Architectural tradeoffs for survivor sequence memory management in viterbi decoders. *IEEE Transactions on Communications*, 41:425–429, 1993.
- [25] David R. Galloway. The transmogrifier c hardware description language and compiler for fpgas. In *Field-Programmable Custom Computing Machines*, pages 136–144, 1995.
- [26] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.
- [27] P. Hilfinger. A high-level language and silicon compiler for digital signal processing. In *IEEE International Symposium on Circuits and Systems*, 1985.
- [28] James C. Hoe. *Operation-Centric Hardware Description and Synthesis*. PhD thesis, MIT, Cambridge, MA, 2000.
- [29] James C. Hoe and Arvind. Synthesis of Operation-Centric Hardware Descriptions. In *ICCAD*, pages 511–518, 2000.

- [30] David Ku and Giovanni DeMicheli. Hardwarec – a language for hardware design (version 2.0). Technical report, Stanford, CA, USA, 1990.
- [31] Sharad Malik. Analysis of cyclic combinational circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems*, pages 950–956, 1994.
- [32] H. Man, J. Rabaey, P. Six, and L. Claesen. Cathedral-ii: A silicon compiler for digital signal processing. *IEEE Des. Test*, 3:13–25, November 1986.
- [33] N. Park and A. Parker. Sehwa: A program for synthesis of pipelines. In *Papers on Twenty-five years of electronic design automation*, 25 years of DAC, pages 595–601, New York, NY, USA, 1988. ACM.
- [34] Daniel L. Rosenband and Arvind. Modular scheduling of guarded atomic actions. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 55–60, New York, NY, USA, 2004. ACM.
- [35] Luc Séméria and Giovanni De Micheli. Spc: synthesis of pointers in c: application of pointer analysis to the behavioral synthesis from c. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design, ICCAD '98*, pages 340–346, New York, NY, USA, 1998. ACM.
- [36] Nir Shavit and Dan Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.
- [37] Byoungro So, Mary W. Hall, and Pedro C. Diniz. A compiler approach to fast hardware design space exploration in fpga-based systems. In *In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 165–176, 2002.
- [38] D. Soderman and Y. Panchul. 1.5: Implementing c designs in hardware: A full-featured ansi c to rtl verilog compiler in action. In *Proceedings of the International Verilog HDL Conference and VHDL International Users Forum*, pages 22–, Washington, DC, USA, 1998. IEEE Computer Society.

- [39] D. Soderman and Y. Panchul. Implementing c algorithms in reconfigurable hardware using c2verilog. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, FCCM '98, pages 339–, Washington, DC, USA, 1998. IEEE Computer Society.
- [40] Charles E. Stroud, Ronald R. Munoz, and David A. Pierce. Behavioral model synthesis with cones. *IEEE Design and Test of Computers*, 5:22–30, 1988.
- [41] SystemC. <http://www.systemc.org/home/>.
- [42] J. E. Thornton. Parallel Operation in the Control Data 6600. *AFIPS Proc. FJCC*, Pt. II, Vol. 26:33–40, 1964.
- [43] Andrew J. Viterbi. Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm. *IEEE Transactions on Information Theory*, IT-13(2):260–269, April 1967.
- [44] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih wei Liao, Chau wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. Suif: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29:31–37, 1994.