A Performance Driven Approach for Hardware Synthesis of Guarded Atomic Actions

by

Daniel L. Rosenband

Bachelor of Science, Computer Science and Engineering
Massachusetts Institute of Technology, 1997

Master of Engineering, Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 1998

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

August 2005

Author .....
Department of Electrical Engineering and Computer Science
August 26, 2005

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

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

A Performance Driven Approach for Hardware Synthesis of Guarded Atomic Actions

by

Daniel L. Rosenband

Submitted to the Department of Electrical Engineering and Computer Science
on August 26, 2005, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Hardware designers are facing new challenges in the design of complex ASIC's and processors as their sizes approach up to 100 million logic gates. We believe no adequate solution exists that allows designers to specify hardware which takes full advantage of the available resources in these devices. The hardware design specification languages are either too low level to support efficient large scale design (for example, Verilog), or the language and synthesis methodology is so high-level that the designer's micro-architectural ingenuity is lost in the design process. This results in circuits that oftentimes do not match the designer's expectations (for example, C-based behavioral synthesis).

This thesis presents a design methodology and related synthesis algorithms that address several of the key issues of hardware design specification and high-level synthesis while avoiding the pitfalls of past approaches. The areas we focus on are modular compilation and performance specification. The modular flow allows for the separate compilation of modules and ensures the correct usage of module interfaces by attaching annotations with well defined semantics to them. We also introduce performance specifications as a core part of a design description. This allows a designer to more easily achieve the expected design performance and it allows for rapid micro-architectural exploration. We chose guarded atomic actions as the foundation of this research because of their clean execution semantics. These semantics allow for easy design transformation (either manual or compiler driven) while ensuring that the correctness of the design is maintained.

We demonstrate the practicality and power of this methodology using several examples, such as a processor which from a single design description can automatically be transformed into an unpipelined processor or a superscalar processor simply by changing a single-line performance specification.

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

Acknowledgments

I would like to thank my advisor Professor Arvind for his support and advice. He is a great mentor and a joy to work with. Many of the ideas in this thesis were developed jointly with him and I appreciate the many hours we spent refining the ideas to obtain more satisfactory results. At a personal level Arvind has always been fun to spend time with. I thank my committee members, Professors Krste Asanovic, Srini Devadas, and Chris Terman for their time and helpful feedback. I would like to thank James Hoe for the many interesting discussions about rule-based synthesis. Nirav Dave and Michael Pellauer were great sounding boards for the ideas in this thesis and were invaluable with their help in using the Bluespec compiler. Rishiyur Nikhil, Jacob Schwartz, Mieszko Lis, and Joe Stoy provided useful feedback during the development of the thesis ideas. Ed Suh and Charles O'Donnell were fun office mates / office neighbors. Our baseball discussions provided good breaks from the thesis work. I thank my parents for their constant love and support throughout my studies. My brother and sister were also always supportive. Special thanks to Jihye Whang for the many good times we have had together.

Contents

Chapter 1 ..... **11**
Introduction **11**
1.1 The designer's dilemma 12
1.2 Why design exploration is important 13
1.2.1 Longest prefix match (LPM) 14
1.2.2 LPM pipelines 15
1.2.3 LPM implementation results 17
1.2.4 LPM lessons learned 17
1.3 Why is design exploration difficult in traditional hardware design flows? 18
1.4 Guarded atomic actions 19
1.5 Thesis contributions 20
1.5.1 Performance specifications and their implementation 21
1.5.2 Modular rule-based synthesis 22
1.6 The failed promise of high-level behavioral synthesis 23
1.7 Thesis outline 24
Chapter 2 ..... **25**
Guarded Atomic Actions **25**
2.1 Guarded atomic action execution model 25
2.2 Guarded atomic action examples 27
2.3 Why guarded atomic actions are useful 29
2.4 Synthesis of guarded atomic actions 30
Chapter 3 ..... **35**
The Modular Rule Language **35**
3.1 The Modular Rule Language (MRL) 36
3.1.1 MRL abstract grammar 36
3.1.2 Rules 38
3.1.3 Interface methods 38
3.1.4 Actions 39
3.1.5 Local bindings 40
3.1.6 Module hierarchy 40
3.1.7 Syntactic sugar 42
3.1.8 MRL vs. Bluespec and ATS 42
3.2 MRL to FRL translation 43
3.2.1 Flattening 45
3.2.2 When lifting 48
3.3 FRL execution semantics 51
3.3.1 Rule execution 51
3.3.2 Sequential execution of rules 54
3.4 Chapter summary 55
Chapter 4 ..... **57**
Modular Compilation **57**
4.1 The goal of modular compilation 59

4.1.1 FIFO interface wiring 61
4.1.2 FIFO interface scheduling 62
4.2 Interface method annotations 64
4.2.1 Conflict matrices 68
4.3 Module hierarchy 70
4.4 Rule scheduling using module interface annotations 72
4.4.1 Rule validity 72
4.4.2 Rule scheduling 73
4.4.3 Rule circuit generation 75
4.5 Deriving module interface annotations 77
4.6 Module compilation 78
4.7 Results 81
4.8 Possible improvements to the modular flow 82
4.9 Chapter summary 84
Chapter 5 85
Performance Specification and the EHR 85
5.1 Understanding scheduling as rule composition 87
5.1.1 Rule composition 87
5.1.2 Rule composition using conditional actions 89
5.1.3 Performance constraints 91
5.2 Transforming composed rules 93
5.2.1 Composition of rules with only register method calls 94
5.2.2 The Ephemeral History Register (EHR) 96
5.3 Modular composition 98
5.4 Performance driven composition algorithm 101
5.5 Specifying schedules for a pipelined processor 103
5.6 Mixed rule and method constraints 109
5.7 Generalizations 111
5.8 Chapter summary 111
Chapter 6 113
Circuit and Performance Evaluation 113
6.1 Pipeline FIFO circuits 114
6.2 Multi-constrained modular composition 118
6.3 Processor and GCD evaluation 120
6.4 Chapter summary 127
Chapter 7 129
Related Work 129
7.1 Guarded atomic actions 129
7.2 Traditional behavioral synthesis 130
7.3 Other efforts 131
Chapter 8 133
Summary and Future Work 133
8.1 Future work 134
Bibliography 137

List of Figures

Figure 1-1: LPM lookup 14
Figure 1-2: LPM algorithm 15
Figure 1-3: LPM pipelines 16
Figure 1-4: LPM results 17
Figure 1-5: Processor pipeline constraints 21
Figure 2-1: Guarded atomic action execution model 26
Figure 2-2: GCD rules 27
Figure 2-3: GCD execution example 27
Figure 2-4: Two stage processor 28
Figure 2-5: Two stage processor rules 29
Figure 2-6: Synthesized guarded atomic actions 30
Figure 2-7: Simple state update arbitration 31
Figure 2-8: Prioritized state update arbitration 32
Figure 3-1: MRL grammar 37
Figure 3-2: MRL naming conventions 38
Figure 3-3: FRL grammar 44
Figure 3-4: The MODMERGE procedure 46
Figure 3-5: Inlining 46
Figure 3-6: The FLATTEN procedure 46
Figure 3-7: Proc / Ctr module merge 47
Figure 3-8: Simple when lifting 49
Figure 3-9: Conditional when lifting 49
Figure 3-10: When lifting transformations 50
Figure 4-1: A modular design 59
Figure 4-2: 2-Element FIFO 60
Figure 4-3: FIFO interface 61
Figure 4-4: Simple use of FIFO module 62
Figure 4-5: FIFO method overlap 63
Figure 4-6: Interface method annotations 67
Figure 4-7: Register annotations 68
Figure 4-8: Three FIFO conflict matrices 69
Figure 4-9: MAKETREE algorithm 71
Figure 4-10: MAKETREE operation 71
Figure 4-11: VALIDRULE procedure 73
Figure 4-12: DeriveRel procedure 74
Figure 4-13: Annotation lattice 74
Figure 4-14: Modular circuit generation 75
Figure 4-15: Derived FIFO annotations 77
Figure 4-16: Modular COMPILE procedure 80
Figure 4-17: Flat vs. modular compilation 82
Figure 4-18: Non-tree module structure 83

Figure 5-1: ARPG syntax 92
Figure 5-2: Method indexing procedure 95
Figure 5-3: The Ephemeral History Register 96
Figure 5-4: EHR conflict matrix 97
Figure 5-5: Method renaming procedure 100
Figure 5-6: Performance driven scheduling algorithm 102
Figure 5-7: PRUNE procedure 103
Figure 5-8: 4-stage processor code 104
Figure 5-9: 4-stage processor pipeline 105
Figure 5-10: Single-element FIFO with bypass 105
Figure 5-11: bE FIFO 107
Figure 5-12: Cache block diagram 109
Figure 5-13: Cache code 110
Figure 6-1: Original FIFO circuit 114
Figure 6-2: Pruned FIFO data register 115
Figure 6-3: No flow control full register (deq.en = deq.rdy) 116
Figure 6-4: Flow-through FIFO circuit 117
Figure 6-5: Flow-through FIFO circuit optimized (1) 118
Figure 6-6: Flow-through FIFO circuit optimized (2) 118
Figure 6-7: Split EHR 120
Figure 6-8: GCD results 121
Figure 6-9: 4-stage processor results 123
Figure 6-10: Component delays 123
Figure 6-11: Moving logic across a mux 125
Figure 6-12: FIFO states after deq and clear operations 126

Chapter 1

Introduction

Hardware designers are facing new challenges in the design of complex ASIC's and processors as their sizes approach 10's of millions or even 100 million logic gates. Some of these challenges exist simply because of the dramatic increase in design size, while others exist due to the shrinking of the physical feature size of the underlying semiconductor technology. Addressing these scaling challenges is important and is continuing to attract substantial attention in the EDA community. However, we believe no adequate solution exists that allows designers to specify hardware that takes full advantage of the available resources in these devices. The hardware design specification languages are either too low level to support efficient large scale design (for example, Verilog), or the language and synthesis methodology is so high-level that the designer's micro-architectural ingenuity is lost in the design process, resulting in circuits that oftentimes do not match the designer's expectations (for example, C-based behavioral synthesis).

This thesis presents a design methodology and related synthesis algorithms that addresses several of the key issues of hardware design specification and high-level synthesis while avoiding the pitfalls of past approaches. The areas we focus on are design re-use (how can we ensure the correct usage of module interfaces), and performance specification (how can we make performance specifications a part of the design description). We demonstrate the practicality and power of this methodology using several examples. For example, we show a

processor which from a single design description can be transformed automatically into an unpipelined processor or a superscalar processor simply by changing a one-line performance specification.

We chose guarded atomic actions as the foundation of this research because of their clean operational semantics. These semantics allow for easy design transformation (either manual or compiler driven) while ensuring that the correctness of the design is maintained. In addition, past work has shown that complex hardware can be conveniently described using guarded atomic actions[3], and that these descriptions can automatically be transformed into hardware[27-29]. In addition, Bluespec Inc. has developed an industrial strength high-level language for rule-based synthesis which facilitated our experimentation[8].

In the next sections we describe more clearly why existing design specification and synthesis solutions are inadequate. We then introduce guarded atomic actions (rules) and show how Hoe and Arvind were able to generate efficient circuits from rule-based descriptions. Next, we describe the thesis contributions and conclude the chapter with an outline for the remainder of the thesis.

1.1 The designer's dilemma

Simply by looking at the numbers, it is clear that hardware design is becoming increasingly complex. In the year 2000 a complex ASIC had roughly 1 million logic gates. Today in 2005, it has roughly 10 million logic gates, and by the year 2010 a complex ASIC will likely have 100 million logic gates. At the same time, due to budget constraints, the design team size must remain constant at 10 to 30 people per ASIC and the design time must not exceed 18 months. Hence, designers must become more productive just to keep up with the design size.

Along with the sheer size of the designs, there are other factors that are stressing the design process. At the physical level, many electrical issues (crosstalk between routes, power distribution, etc.) are becoming relevant and require new tools and iterations in the design flow. At the front-end of the design process, which we focus on in this thesis, a single designer must now design blocks with 1 million or more logic gates—blocks that are systems themselves. As a result, whereas a designer used to receive a mostly complete micro-architectural specification from an architect, designers must now develop their own complex interfaces, choose data structures and algorithms, and develop the block's micro-architecture. This means that

designers now have dramatically more work to do than simply "coding up" a larger block. Hence, their workload is increasing by more than 2x every 18 months.

There are three ways a designer can satisfy this increased workload:

(i) The design flow improves.

(ii) The designer gets "better".

(iii) The designer cuts corners by making conservative (easy to implement) but wasteful (area, performance, power, etc.) design choices without exploring alternatives.

We believe most of the improvement in design productivity has been achieved by (ii) and (iii) over the past 10 years, and is increasingly achieved via conservative and not well thought out design (iii). The reason for this is that the front-end of the design flow (specification, verification, and synthesis) has not changed substantially in this time frame and its use has matured—designers will not become much more efficient at writing RTL Verilog. (Clearly, the design tools themselves have improved to handle larger designs, a big challenge in itself, but the flow has remained mostly constant.)

Relying on ever more conservative and wasteful design is not an attractive prospect for improving productivity of hardware designs. Much ingenuity and potential is being wasted by not allowing designers the flexibility to experiment with micro-architectures, not providing the infrastructure to incorporate complex data structures into the design, and not providing mechanisms to easily re-use both mundane and complex blocks. As a result, market demands for low power, low cost and high performance ASIC's are not fully satisfied. This is the motivation for our research on high-level synthesis, the goal of which is to allow the designer to take advantage of the tremendous resources that large semiconductors provide.

1.2 Why design exploration is important

As previously mentioned, we believe that design exploration is an important part of the design process that is falling by the wayside due to limitations in the traditional RTL design flow as well as due to the severe time constraints in the design process. A contribution of this thesis is to enhance our ability to experiment with alternate designs—either by allowing modules with different performance characteristics to be easily and *safely* swapped in and out of a design, or

by allowing the designer to easily trade-off such factors as cycle time and throughput via performance constraints.

To motivate this aspect of the design process we present a small case study[2]. This should both enhance the claims on why the design flow needs to change and it will also justify some of the work we present in later chapters. This example will also be used to highlight why traditional behavioral synthesis is not the correct approach to improving design productivity.

1.2.1 Longest prefix match (LPM)

Longest prefix match (LPM) is a key hardware component in high-end IP routers[22]. The basics of the problem are: given a 32-bit IP address (IPA) and a table of address / route pairs, return the route corresponding to the table entry with the longest matching address prefix. Any reasonable implementation must be pipelined (throughput is a major driver in this problem), and must utilize off-chip memories (the tables are too large to store on-chip). This is illustrated in Figure 1-1.

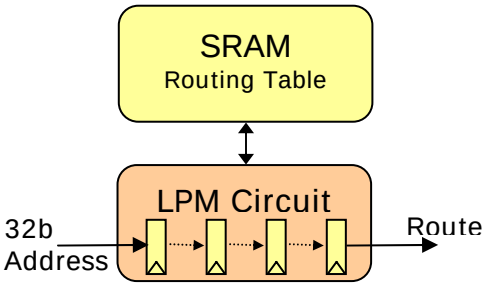


Figure 1-1: LPM lookup

Many complex algorithms have been developed to optimize the throughput and latency of the longest prefix match problem. Most of these algorithms trade off the compactness of the table representation in the SRAM with the number and width of the memory accesses. In comparison to state-of-the-art lookup algorithms, the lookup procedure used for this study is simplistic, but suitable to illustrate the challenges facing hardware designers. (Understanding the details of the algorithm is not required to understand the points we will make about the resulting hardware.)

The basic idea behind the lookup algorithm (see Figure 1-2) is to store the lookup table as a tree data structure. Starting at the root, each non-leaf node contains a table that points to

the appropriate node at the next level in the tree. These tables are indexed using one of three sections of the IP address. Hence, each lookup requires up to three memory references depending on how soon a leaf node is encountered—leaf nodes contain the desired route information:

```
int LPM(IPA ipa)  {
   int p;

   /*** first memory reference ***/
   p = SRAM [rootTableBase + ipa[31:16]];
   if (isLeaf(p))
      return p;

   /*** second memory reference (if required) ***/
   p = RAM [p + ipa [15:8]];
   if (isLeaf(p))
      return p;

   /*** third memory reference (if required) ***/
   p = RAM [p + ipa [7:0]];
   return p; // must be a leaf
}
```

Figure 1-2: LPM algorithm

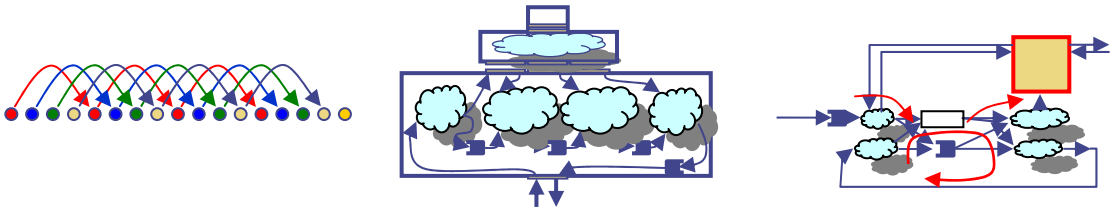
1.2.2 LPM pipelines

The key constraint in implementing this algorithm efficiently is that it must provide high throughput. Because the external memories usually have a read latency of at least 4 cycles, this means that the design must be pipelined and multiple lookups must occur simultaneously. There are multiple ways that such pipelining can be performed. We illustrate three of them in Figure 1-3:

a) Static pipeline: each lookup is statically assigned a time when it accesses memory. If the memory latency is 3, then lookup 1 accesses memory on cycles 0, 3, and 6; packet 2 accesses memory on cycles 1, 4, 7; packet 3 on cycles 2, 5, 8; and packet 4 on cycles, 9, 12, and 15. This is the implementation that many designers would prefer because of its static nature and simplicity. It has the drawback that memory bandwidth, and hence throughput, is wasted since some lookups will not require three memory references.

b) Dynamic pipeline: each lookup only performs the memory references that are required. By using FIFO's between lookup stages we achieve elasticity in the pipeline and hence higher throughput than in the static case. A drawback is that the FIFO's require more state than the static pipeline to support optimal throughput.

c) Circular pipeline: Addresses rotate through the lookup state machine until the destination route is found. The result is then placed in a completion buffer so that the results can be returned in the correct order. This design achieves the same throughput as the dynamic pipeline since memory bandwidth is dynamically assigned to the addresses that require additional memory references (those that do not require more memory references would already have been placed in the completion buffer).



(a) static **(b) dynamic** **(c) circular**

Figure 1-3: LPM pipelines

All three of these pipelines are reasonable. However, we believe that most designers would pick the static pipeline (a) or the dynamic pipeline (b) as the design of choice. The static design would be chosen for its perceived simplicity because of its static nature, while the dynamic pipeline would be chosen for its improved throughput. The circular pipeline contains a more complicated architecture and implementing a completion buffer correctly can be challenging. However, the circular pipeline has the advantage of being the most robust design with respect to changes in the lookup algorithm, changes in memory latency, etc.

Precisely what the trade-offs for area, timing, and throughput are cannot be determined unless the designs are actually implemented. It should be obvious that designers will be faced with many similar choices when designing logic blocks with over one million gates, except that the stakes are orders of magnitude higher in such cases. Since we find surprises in the implementation of these simple LPM pipelines, designers would likely find numerous surprises if they took a close look at many of their larger blocks.

1.2.3 LPM implementation results

Figure 1-4 shows implementation results for all three LPM pipelines. All designs, except for Static 2, were implemented by two different designers in two different design languages (Verilog and Bluespec). We show only one set of numbers for each design since the variation between the results for each pair of designers was less than 10%.

LPM Pipeline	Area (gates)	Speed (ns)	Memory Utilization (%)
Static	8,898	3.60	63.5
Static 2	2,391	3.32	63.5
Dynamic	15,910	4.70	99.9
Circular	8,170	3.67	99.9

Figure 1-4: LPM results

Two of the results were surprising. First, the circular pipeline turned out to be substantially more area efficient than the dynamic pipeline. The reason for this was that the area overhead of the FIFO's in each stage of the dynamic pipeline could be aggregated in the completion buffer. Second, we were surprised that the static pipeline was not substantially smaller than the other designs—given its simple architecture we expected a low gate count. After asking a third designer to implement the static pipeline we obtained substantially better results—an almost 75% reduction in gate count (Static 2). The reason for this reduction in gate count was that rather than using a separate state machine for each active lookup, the state machines could actually be shared among the simultaneously occurring this—this was a micro-architectural optimization.

1.2.4 LPM lessons learned

The results of this case study confirm two insights:

- Micro-architecture drives the performance (area, timing, etc.) of a design.

- Making it easier to experiment with architectures to obtain realistic area, timing, and performance numbers is a key component of any specification and synthesis framework for next-generation ASIC's.

The first point may seem trivial. However, it is often ignored when studying hardware synthesis as the focus is usually on how one language compares to another language when implementing a given micro-architecture. These differences are usually in the single digit percentage range, much smaller than changes between micro-architectures. In this small example we had a variation of more than 6x in area, 30% in timing and 35% in throughput. One can only imagine how significant these numbers become in much larger blocks.

The second insight is a consequence of the fact that micro-architecture is so important. It states that a key component of any new synthesis and specification system must make it easier to implement and experiment with micro-architectures. For this to happen, advances are required in two dimensions: (i) it must become easier to specify a micro-architecture and (ii) changing the micro-architecture of part of the design, for example by adding a pipeline stage or by swapping in a high performance module for a lower performing one, should not break the rest of the design. This thesis contributes in both of these dimensions.

1.3 Why is design exploration difficult in traditional hardware design flows?

A traditional RTL design flow requires a designer to schedule all pipelines and resources before coding begins. The designer must not only be aware of the scheduling, but must also implement it—this means coding the scheduling state machines, implementing arbitration circuits to shared resources and coding the multiplexer (mux) logic that ensures the correct values are written to each state in every cycle. This process has the advantage of giving the designer full power over implementation details. Generally it also ensures that throughput and latency performance expectations are met since the designer carefully crafted the scheduling logic.

The disadvantage of this approach is that the scheduling logic becomes deeply entwined in the functional part of the design. This leads to verification challenges because of the difficulty in identifying whether mistakes were made in the scheduling or functional logic. More interesting for this thesis, the process also makes the design rigid with respect to design

modification and makes design exploration impractical. Adding a pipeline stage because cycle times where not met, replacing a memory with another memory that has larger latency, or changing the access priorities of a shared resource often has a ripple effect through the entire design. Any of these changes require modifications to the schedule and often a substantial effort to modify the corresponding logic. As a result, designers strive for conservative design so that they are unlikely to have to make changes at a later stage. Design exploration as we advocated in Section 1.2 is rarely considered due to the effort involved in making the required changes.

Often the only time design changes are considered is in the synthesis or physical design process. If timing closure is posing substantial problems then every effort is first made to restructure combinational logic to reduce the critical path. Such changes tend not to alter the scheduling logic and are less error-prone than, for example, the restructuring of a pipeline. Only if timing can absolutely not be met via combinational logic changes are pipeline changes considered. Because of the effort involved, these often then lead to delays in the chip design.

1.4 Guarded atomic actions

This thesis builds on guarded atomic actions as a foundation. It is a design style that is quite different from traditional RTL design and has the potential to address the shortcomings of the RTL design process. Guarded atomic actions, which we also refer to as rules, have been used for decades in the form of asynchronous languages to describe distributed algorithms[10, 33]. Some of the examples in the hardware domain are Dill's Murphi[16], Straunstrup's Synchronous transactions[51], Sere's Action systems[43], and Arvind & Shen's TRS's[3, 50]. The main idea underlying all such descriptions is that any hardware system has a (structural) state component that can be captured by a set of variables that represent registers or storage, and the behavior is nothing but a set of rules, that is atomic actions with guards, on this state. A precise and useful semantics emerges from the fact that any legitimate behavior of the system can be understood as a series of atomic actions on this state.

The key difference between this design style and traditional RTL is that a schedule of rule executions need not be specified by the designer. Instead, designs are constructed such that the design is functionally correct for *any* order of rule execution. In the context of a hardware design, this means a designer can focus on individual hardware components without

worrying about interactions with other parts of the design. For example, a rule could be used to represent a pipeline stage, or even the logic to execute a particular instruction in a pipeline stage. This rule would describe the behavior of the pipeline stage in isolation and would not need to address what happens if the previous or following stages execute simultaneously. The reason that such an abstraction is possible is that the behavior of any execution must be explainable as the sequential and atomic execution of each rule.

Almost by definition, it is easier to create functionally correct designs using guarded atomic actions than using traditional RTL because the entire rule-based description focuses on functionality. In contrast, RTL contains a mix of functionality and scheduling. This focus on functionality along with the operational semantics of rule-based descriptions also makes them amenable to formal verification.

Up until recently, a major drawback has been that efficient circuits could not be generated from rule-based descriptions. The primary reason for this is that any reasonable hardware requires many components to execute in parallel. However, parallel execution appears to contradict the requirement that rule execution must appear to occur sequentially. Hoe and Arvind[27-29] were able to solve this problem by generating circuits that allow multiple rules to execute concurrently within each clock cycle while maintaining the appearance of sequential execution. The Achilles heel in this process is that the designer relies on a compiler to derive a scheduler that executes a sufficient number of rules in each cycle. If the compiler does not find the expected parallelism then the designer has had only unattractive solutions to fix the problem.

In summary, for designs where the compiler derives sufficient parallelism guarded atomic actions present an attractive model for hardware design. By focusing on functionality in each pipeline stage rather than on the scheduling logic details a designer is able to more easily refine a design to add functionality or satisfy timing constraints. Design exploration using rules is easier than traditional RTL for the same reason.

1.5 Thesis contributions

The two main thesis contributions are a modular rule-based synthesis flow and a performance driven synthesis flow that allows a designer to specify which rules should execute simultaneously in each cycle. We describe these contributions in the following subsections.

1.5.1 Performance specifications and their implementation

As previously mentioned, the motivation behind this thesis was to improve the design methodology and synthesis algorithms for large semiconductors. Guarded atomic actions have many attractive attributes that we believe makes them a good candidate for large scale hardware design. However, as outlined, several key problems exist in the methodology. The primary problem has been that the designer cannot control the scheduling process, leading to unpredictable and at times unacceptable performance (throughput). This thesis presents new synthesis algorithms that solve this problem. The basic idea behind the algorithms is that the designer should write the rules as before but can now also include a performance specification. The performance specifications specify which rules should execute concurrently within a cycle and what order they should appear to execute in. This allows a designer to precisely specify what the scheduling for a given micro-architecture should be without needing to explicitly code the scheduler, the mux's, etc., as would be required in a traditional RTL flow.

An example of the use of performance specifications is a processor pipeline. Assuming rules F (fetch), D (decode), E (execute), M (memory), and W (write back) describe their respective pipeline stages, a designer could first synthesize and simulate the design to verify that the functionality is correct. The designer would then examine the performance of the circuits. In Hoe and Arvind's synthesis framework it is possible that only the rules corresponding to alternating pipeline stages can execute together within a cycle. Such a circuit remains functionally correct since the processor still executes correctly, but is clearly unacceptable from a performance standpoint. In the synthesis flow proposed in this thesis, the designer feeds the original, unaltered, processor description along with performance constraints into a compiler. For the three constraints shown in Figure 1-5 the compiler would generate (a) an unpipelined processor, (b) a pipelined processor in which all stages can execute concurrently, and (c) a superscalar processor in which two instructions can concurrently execute in each stage.

```
a)  F < D < E < M < W
b)  W < M < E < D < F
c)  W < W < M < M < E < E < D < D < F < F
```

Figure 1-5: Processor pipeline constraints

This methodology provides the benefits of rule-based design—the focus of the design description is functionality rather than scheduling logic—while maintaining the ability to control the scheduling such that a designer's intent is not lost in the design process. The high-level performance specifications also allow a designer to experiment and change the scheduling more rapidly than is possible in traditional RTL design.

1.5.2 Modular rule-based synthesis

The second major contribution of this thesis is a modular compilation flow for a rule-based synthesis system. The challenge in this part of the thesis is to create an abstraction that allows rules to interact with modules while maintaining their atomic and sequential semantics. We achieve this by introducing a set of interface method annotations that specify how methods interact. The annotations provide sufficient information to determine whether two rules that call a module's methods can be scheduled to execute concurrently while maintaining the appearance of executing sequentially and atomically. We also present a compilation algorithm that shows how annotations can be propagated through a module hierarchy to derive the annotations for higher-level modules.

This modular compilation flow is important for several reasons. In the context of rule-based synthesis, one of the values of the modular flow is that it makes the design flow scalable and capable of handling larger designs. A broader contribution is that the modular flow presents an attractive model for design reuse and intellectual property (IP) exchange. By attaching scheduling annotations to module interfaces we introduce constraints on how a module can be used, for example that the FIFO enqueue and dequeue methods must not be called simultaneously. A compiler then ensures that these constraints are not violated. This contrasts with traditional IP exchange in which a designer must read through a document and manually ensure that the block is used correctly.

Both the modular compilation and performance specification contributions simplify the design experience. The technical link between them is that the performance specifications rely on the module annotations. In the modular flow, annotations are derived to describe a module's behavior. In the performance specification flow, the designer specifies constraints using exactly the same type of annotations and the compiler transforms the design to satisfy

these constraints. The modular synthesis algorithms can then be used to compile the resulting design.

1.6 The failed promise of high-level behavioral synthesis

In this section we briefly review how the framework of this research differs from traditional behavioral synthesis. We discuss related work at the end of the thesis but briefly review traditional behavioral synthesis in this section since it is most closely related.

High-level behavioral synthesis has been proposed as a solution to help designers produce designs of ever increasing sizes—precisely the problem this thesis targets. Approaches have used new specification languages ranging from behavioral Verilog[32], to C[19], to SystemC[41, 53]. These languages themselves are far richer than traditional RTL languages (Verilog and VHDL) and hence were assumed to hold promise in alleviating the design process. However, we believe the major reason for these tools' failure among designers is their attempt to automatically infer micro-architectures.

The LPM problem from Section 1.2 illustrates why traditional behavioral synthesis did not succeed. In a behavioral flow the designer would write the LPM procedure, as written in Figure 1-2. The behavioral synthesis tool would then infer the state, data paths and control logic to implement the procedure. An advanced tool would perhaps also pipeline the design. But which pipeline would it choose? How much state does it infer? What will the resulting throughput be? All these questions are unknowns before the synthesis tool is run. Additionally, there are insufficient mechanisms to direct the synthesis process, for example to choose the static pipeline as opposed to a dynamic pipeline. Hence, the designer is rolling dice in this process and hoping that the tool chooses a "good" implementation. If the outcome is not as desired, there is little the designer can do to direct the implementation.

In contrast to traditional behavioral synthesis approaches, our philosophy has been not to preempt the ingenuity of the designer, especially when it comes to choosing a micro-architecture. Our goal is to provide the designer the mechanisms to easily create and experiment with architectures of his or her choosing.

We should note that behavioral synthesis tools have been successful at optimizing computational data paths in DSP style designs. They are very good at taking a control data-flow graph (CDFG) for DSP style computations[18, 19, 23, 32] and transforming the graph to

optimize throughput, latency, area, etc. However, these algorithms become less effective when they do not control the entire schedule and need to interact with external components, for example the memory in the IP example. In addition, CDFG synthesis tools generally do not handle dynamic design properties efficiently because they create static schedules for the design. We saw in the LPM example that a static schedule is not necessarily the optimal design choice.

It is our belief that DSP-style design is important but that it represents only a small sub-set of the design space. Our focus is on allowing the designer to more efficiently express designs that contain a mix of data paths, state machines, and complex control logic, something CDFG compilation does not handle efficiently.

1.7 Thesis outline

The next chapter presents an overview of guarded atomic actions and the synthesis algorithms that Hoe and Arvind developed for them. The chapter is a review to assist the reader in becoming familiar with guarded atomic actions. Chapter 3 presents a new modular rule-based language (MRL) and an operational semantics that specifies how MRL must behave. Chapter 4 then introduces a modular synthesis flow that shows how to generate hardware from MRL programs. A key contribution in this chapter is a set of interface scheduling annotations that specify how a module can be used. Chapter 5 presents a new scheduling algorithm that allows a designer to specify performance constraints. A synthesis algorithm accepts the constraints and the original design as input and produces as output a design that satisfies the performance constraints and is also guaranteed to be functionally equivalent to the original. Chapter 6 examines and evaluates the circuits that are produced by the synthesis algorithms from Chapter 5. In Chapter 7 we discuss related work, and conclude in Chapter 8 with a brief summary of the thesis.

Chapter 2

Guarded Atomic Actions

This thesis uses guarded atomic actions as a foundation to build on, primarily because of their clean semantic model, but also because of Hoe and Arvind's initial successes in synthesizing efficient logic from their descriptions. This chapter presents a review of guarded atomic actions: their operational semantics, their use, their benefits, and the basics of Hoe's and Arvind's synthesis algorithm.

2.1 Guarded atomic action execution model

Each atomic action (or rule) consists of a body and a guard. The body describes the execution behavior of the rule if it is enabled. The guard (or predicate) specifies the condition that needs to be satisfied for the rule to be executable. We write rules in the form:

```
rule Ri: when πi(s) =>
            s := δi(s);
```

Here, π_i is the predicate and $s := \delta_i(s)$ is the body of rule R_i. Function δ_i is used to compute the next state of the system from the current state s.

The execution model for a set of rules is to non-deterministically pick a rule whose predicate is true and then to atomically execute that rule's body. The execution continues as long as some predicate is true:

```
while (some π is true) do
   1) select any Ri , such that πi(s) is true
   2) s := δi(s);  // update the state
```

Figure 2-1: Guarded atomic action execution model

We often refer to this as the *atomic and sequential* execution model because atomicity and sequential execution are its two key properties. By atomic execution we mean that a rule can never appear to execute partially. Hence, the state of the system should only be observed either before the rule begins executing or after it completes execution. By sequential execution we mean that it must appear that rules execute in some sequential order. This means that a rule must observe all state updates that rules earlier in the sequence performed. Similarly, a rule must not observe any of the state updates that rules later in the sequence perform. We provide a more formal definition of this model in the next chapter.

A property of the guarded atomic action execution model is that rules do not always execute when their guards (predicates) are satisfied. For example, suppose we are given the two rules R_1 and R_2 below and the initial state of the system is *x = 0, y = 0, ctr = 0*.

```
R1: when (x == 0) =>
        x := x + 1;

R2: when (x == y) =>
        ctr := ctr + 1;
```

Both rules' predicates are initially true. Thus, either rule can execute first. After executing rule R_1 we obtain the state: *x = 1, y = 0, ctr = 0*. At this point, rule R_2 has been disabled since its predicate is no longer true. Hence, R_2 cannot execute after a single execution of R_1. If we had chosen R_2 to execute first, we would obtain the state: *x = 0, y = 0, ctr = 1*. At this point both rules' predicates are still true and we could choose either rule to execute next. Thus, rules do not always execute if their guards are true and the behavior of the system can depend on the order of rule execution. In general, although we do want this capability, we discourage a design style in which behaviors vary depending on the order of rule execution. Most designs that we discuss contain rules whose predicates can be simultaneously true. However, the final state in these systems will be same regardless of rule execution order.

2.2 Guarded atomic action examples

This section presents two examples of using guarded atomic actions to describe hardware. The first example computes the greatest common divisor (GCD) of two numbers. The second example contains a portion of a simple processor design.

The following two rules compute the GCD of two numbers *x* and *y* using Euclid's GCD algorithm. The result of the computation is located in register *x* when *y* contains the value 0:

```
Rsub: when ((x >= y) & (y != 0)) =>
        x := x - y;

Rswap: when ((x < y) & (y != 0)) =>
        x, y := y, x;
```

Figure 2-2: GCD rules

An execution example for these rules, given initial values *x = 15* and *y = 6* is shown in Figure 2-3. In this example the application order of rules is deterministic since the two rules' predicates are mutually-exclusive (*x* cannot be both "less than" *y* and "greater than or equal" to *y*). Hence, in each step the rule whose predicate is true is applied to the state of the system (*x* and *y*).

Step #	Rule	x	Y
0	Initial Values	15	6
1	R_{sub}	9	6
2	R_{swap}	3	6
3	R_{sub}	6	3
4	R_{sub}	3	3
5	R_{swap}	0	3
6	Done: Result = 3	3	0

Figure 2-3: GCD execution example

A key difference between these rules and traditional RTL (for example, Verilog) is that both GCD rules modify the same state (*x*) without explicitly arbitrating for access to the register. Instead, any compiler is required to ensure atomic execution of each rule when generating hardware (or software) that implements these (or any other) rules.

Next we show how to design a simple two-stage processor using guarded atomic actions. As shown in Figure 2-4, the processor contains the usual state elements: program counter (pc) and register file (rf). It also contains a FIFO (bu) as the pipeline stage.

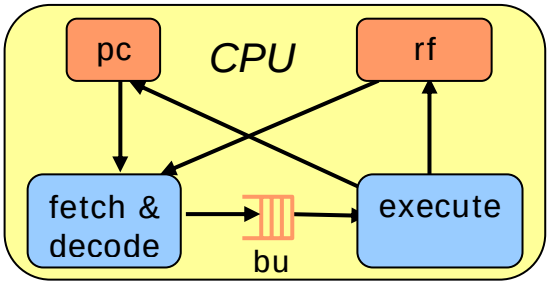


Figure 2-4: Two stage processor

Figure 2-5 shows the processor rules. They are divided into two groups: fetch and decode rules (FD*) and execute rules (E*). The asynchronous (decoupled) and non-deterministic nature of rule-based design is exhibited by the fact that the two stages (FD and E) are completely decoupled, except for their interaction via the *bu* FIFO. So long as the FIFO is not full and does not contain an instruction that writes to a register source of the instruction in the FD stage, the FD rules can execute. Similarly, the E rules can execute whenever the *bu* FIFO is non-empty. Hence, neither set of rules needs to interact directly with the other set of rules. (Note: full / empty status is implied by the *enq* and *deq* FIFO method calls.)

It is also worth pointing out that unlike in the GCD example, the processor rules can execute in many different (non-deterministic) orders, provided that the size of the *bu* FIFO is greater than one. For example, two FD rules can execute in sequence, followed by the execution of 2 E rules in sequence. Or, the FD and E rules could execute in alternating order. At first this might appear to make the design process more difficult since the designer cannot be certain in what order events will occur. However, in many cases[3, 52], the non-deterministic scheduling makes it possible to prove properties about the design as well as refine the design through design transformations. The decoupled nature of the descriptions and possibly non-deterministic scheduling of rules also adds robustness to the design process since a change of the scheduling in one part of the design by definition will not affect the functionality of the rest of the design. A major contribution of this thesis is showing how to maintain this robustness while allowing the designer to also specify desired performance characteristics to direct the scheduling of rules.

```
FDadd:      when ((iMem[pc] == Add{rc, ra, rb}) &
                    !bu.find(ra) & !bu.find(rb)) =>
                bu.enq(EAdd{rc, rf[ra], rf[rb]});
                pc := pc + 1;

FDbz:       when ((iMem[pc] == Bz{rc, addr}) &
                  !bu.find(rc) & !bu.find(addr)) =>
                bu.enq(EBz{rf[rc], rf[addr]});
                pc := pc + 1;

Eadd:       when (bu.first() == EAdd{rc, va, vb}) =>
                rf[rc] := va + vb;
                bu.deq();

Ebztaken:   when ((bu.first() == EBz{vc, va}) & (vc == 0)) =>
                pc := va;
                bu.clear();

Ebznotake:  when ((bu.first() == EBz{vc, va}) & (vc != 0)) =>
              bu.deq();
```

Figure 2-5: Two stage processor rules

Similar to the GCD case we again have multiple rules that modify the same state (the *pc* register, and *bu* FIFO). The designer does not need to worry about how accesses by different rules interact since the execution semantics ensure that each rule is applied atomically to the state of the system. We believe this is one of the major advantages of rule-based synthesis since it allows the designer to ignore the details of this error-prone arbitration logic.

2.3 Why guarded atomic actions are useful

Before discussing efficient hardware generated from rule-based descriptions we should summarize why we believe rule-based descriptions are an attractive model for hardware generation. The key advantages are:

- The design style is asynchronous / decoupled. This makes designs robust with respect to scheduling changes in other parts of the design. For example, rules representing the processor pipeline stages could be written without regard to how they interact with the simultaneous execution of rules in other pipeline stages.
- Designs need not specify the details of arbitration for access to shared state by multiple rules. For example, the two stage processor rules FDadd and Ebztaken

both modify the PC. However, no explicit logic to arbitrate the access to this state needed to be expressed.

- Guarded atomic actions have simple and well defined execution semantics. This makes proving properties about a design and transforming the design possible.

2.4 Synthesis of guarded atomic actions

There is a straightforward translation from rules into hardware. Assuming all state is accessible (no port contention), each rule's π and δ expressions can be easily implemented as combinational logic. As shown in Figure 2-6, a hardware scheduler and control circuit then needs to be added so that in every cycle the scheduler dynamically picks one δ function whose corresponding π condition is satisfied. An arbitration circuit then updates the state of the system with the result of the selected δ function. In this circuit, the φ signals are used to indicate which rule is active. Figure 2-7 shows the arbitration logic for each state element: it takes as input the new state value from each δ function for each piece of state and selects the next state value depending on which rule is active. (If a rule does not change a particular state, then the next state value for that rule/state pair is not meaningful. Hence we would disable state-updates for that rule/state pair.)

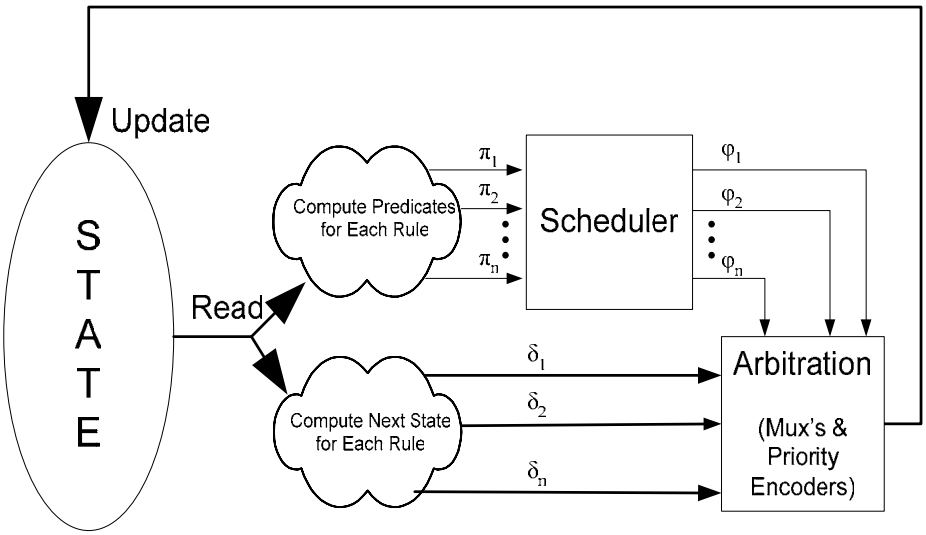


Figure 2-6: Synthesized guarded atomic actions

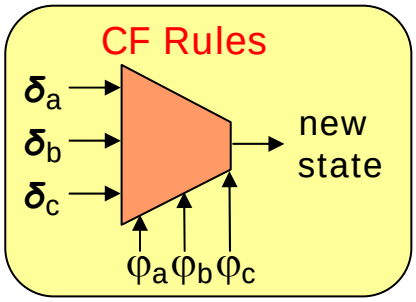


Figure 2-7: Simple state update arbitration

The cycle time in such a synthesis is determined by the slowest π and the slowest δ functions. However, although correct, such an implementation has unsatisfactory throughput because it executes only one rule per cycle. In the processor pipeline from Section 2.2 this would be unacceptable since the designer would expect any reasonable implementation to execute the two processor stages concurrently. Fortunately, it is often possible to execute several rules simultaneously such that the result of the execution matches an execution in which the selected rules are applied in some sequential order—as the semantics of rule execution require. Thus, the challenge in generating efficient hardware from sets of atomic actions is to generate a scheduler which in every cycle picks a maximal set of rules that can be executed simultaneously. We should note that past work and this thesis assumes that each rule executes within a single cycle but implementations where the execution of a rule may stretch over multiple cycles might be an attractive area to investigate.

Both Staunstrup[51] and Hoe[27-29] improved on the above base-line implementation by making the observation that two rules can execute simultaneously if they are "conflict free" (CF), that is, they do not update the same state and neither updates the state accessed (i.e., "read") by the other rule. An example of two CF rules is:

```
R1: when (True) =>
        x := x + 1;

R2: when (y < 7) =>
        y := y + 1;
```

Only the scheduler in the circuits of Figure 2-6 needs to change to support the simultaneous execution of CF rules. Rather than select only one rule at a time (set one δ to true), the

scheduler can now select multiple rules (δ's) to be true, provided that their corresponding predicates (π's) are true and that they are all mutually conflict free.

Arvind and Hoe further observed that two rules (R_1 and R_2) can execute simultaneously if one rule (R_2) does not read any of the state that the other rule (R_1) writes. In this case simultaneous execution of R_1 and R_2 appears the same as sequential execution of R_1 followed by R_2. For this to hold R_2 writes must take precedence over writes to the same state by R_1 and the execution of R_1 must not disable R_2. Such rules are called "sequentially composable" (SC) in[12]. An example of two SC rules is shown below:

```
R1: when (True) =>
        x := y + 1;

R2: when (y < 7) =>
        y := y + 1;
```

Given these two rules and an initial state *x = 0, y = 1*, applying the rules in sequence R_1 followed by R_2 produces the values *x = 2, y = 3*. This is precisely the value we obtain if we apply the above mentioned circuit generation technique.

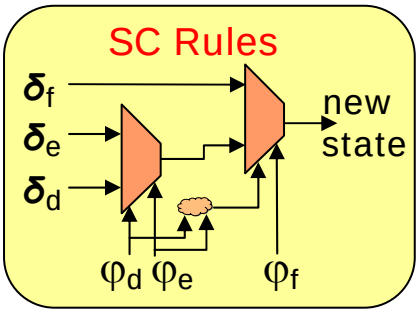


Figure 2-8: Prioritized state update arbitration

To add SC to the circuit of Figure 2-6 we again need to update the scheduler to now also enable sets of rules that are pair-wise (and in a consistent order) SC. The arbitration circuits must now also give priority to rules depending on their SC relationship as shown in Figure 2-8.

Hoe and Arvind showed how to generate a scheduler that selects a maximal subset of applicable rules within each cycle. By using the CF and SC properties they ensured that the outcome of a scheduling step could be explained as atomic firing of rules in some sequence. Their synthesis system supported registers, FIFO's and register files as primitive state elements.

It is important to note that this scheduling process is not user driven. A compiler is automatically deciding which subset of rules is "best" to execute in each cycle. Since no clear heuristic exists to choose the "best" subset, the approach used thus far has been to assign fixed priorities to rules and to have these priorities help guide the compiler in choosing the most appropriate rules to execute. In Chapter 5 we introduce a new scheduling algorithm that allows for more parallelism than Hoe and Arvind were able to derive and also allows the designer to more precisely specify what rules should execute in each cycle.

An important observation is that neither CF nor SC scheduling substantially changes the cycle time of the base-line circuit implementation. The reason for this is that the only logic changes between these implementations lies in the scheduler circuit and the state update arbitration circuits. The scheduling circuit is generally small compared to the rest of the logic and does not impact the cycle time unless the delay of the predicate computation (π) is comparable to the delay of the update function computation (δ). The state update arbitration logic does lie on the critical path. However, the CF style mux is required for even single-rule at a time execution since regardless of whether rules execute simultaneously, the next state value must be chosen from multiple possible sources. Thus, CF arbitration does not increase the critical path of the design over a base-line single-rule at a time implementation. The SC arbitration logic has a longer propagation delay than CF arbitration because the mux's must be staggered to implement a priority encoder—"later" rules must take precedence when updating state. Usually, this additional delay has an impact on cycle time, but is small compared to the computation in the π and stages δ stages. In most cases, prioritized access would have to be arbitrated in a traditional RTL design style as well. Hence, SC circuits are often as efficient as RTL implementations. (Note: the synthesis flow treats the scheduler and arbiter as a single combinational block to allow optimizations across both blocks.)

Another important observation is that Hoe and Arvind's synthesis algorithms do not support the forwarding of values from one rule to another. This means that the values written by one rule cannot be read by another rule within the same cycle. As we will see in Chapter 5, this is a limitation that causes many designs to be scheduled with insufficient parallelism.

Chapter 3

The Modular Rule Language

This chapter introduces a modular rule-based language (MRL) and provides an execution semantics that specifies the behaviors that any implementation of an MRL program must adhere to. We use MRL as the specification language for examples throughout the remainder of the thesis, and the goal of the synthesis algorithms that we introduce in later chapters is to synthesize hardware descriptions written in this language efficiently.

The MRL language can be considered the core of the much richer Bluespec language[4, 8], similar to Hoe and Arvind's ATS as the core of their TRS framework[27-29]. MRL adopts Bluespec's notion of a module which can contain local state elements, interface methods which allow other modules access to its state, and rules which describe the module's internal behavior. The key difference between this framework and Hoe's environment is that MRL supports a user-defined module hierarchy whereas Hoe was limited to synthesizing rules that interact with only a small set of primitive state elements. The difference between MRL and Bluespec is that MRL contains only the constructs that make the scheduling and inter-module communication part of Bluespec a challenge—the part that constitutes the core of the synthesis algorithms. MRL does not contain Bluespec's sophisticated type system, it does not support local functions, does not contain loops, etc. In essence, MRL is an intermediate form of Bluespec after all preprocessing and type checking has been performed, but before any scheduling and module synthesis has begun.

One of the values of this chapter is that it introduces a language that we can use for modular and performance driven synthesis in the following chapters. Another important contribution is that it defines how a modular rule-based language should behave. Bluespec Inc. had developed a modular language before we began this work, but neither a true modular synthesis flow existed, nor were the semantics of the language clear. Given the importance of the guarded atomic action execution model, expressing the semantics of a modular environment is important if we are to use a modular language to describe large-scale designs based on guarded atomic actions.

We begin the chapter by introducing MRL and the ideas behind it. We then explain the execution semantics in two steps: (i) we show how to translate MRL descriptions into a flat rule-based design (FRL) that closely matches the ATS framework in which Hoe and Arvind worked, and (ii) we provide sequential execution semantics for the derived FRL program.

3.1 The Modular Rule Language (MRL)

At a high-level, each MRL program contains a module hierarchy in which each module consists of (i) local state elements (module instances), (ii) local bindings (combinational logic), (iii) interface methods which allow other modules' rules or methods to access the module's internals, and (iv) rules, which define the module's internal behavior. The behavior of any such program can still be explained as a sequential execution of rules. However, rules may be located in many modules and their behavior is expressed via calls to module interface methods that provide access to modules' internal state elements. This contrasts with traditional rule-based descriptions in which all rules are located in a single module and rules interact with primitive state elements only.

3.1.1 MRL abstract grammar

Figure 3-1 shows the grammar of the MRL language. The next subsections discuss each of the language structures and their meaning. We use the following conventions in the grammar:

```
<E>   ≡  1 occurrence of entity of type E
{E}   ≡  0 or 1 occurrence of entity of type E
[E]   ≡  0 or more occurrences of entity of type E
```

```
Program ::=
    [Module Definition]
    [Module Instance]

Module Definition ::=
    module <Module Definition Name>
          [Module Instance]
          [Local Binding]
          [Read Method]
          [Action Method]
          [Rule]
    endmodule

Module Instance ::=
    <Module Definition Name> <Module Instance Name>

Local Binding ::= <Variable> = <Exp>

Read Method ::=
    method <Read Method Name> ([Variable]) =
          return <Exp>
    when <Exp>

Action Method ::=
    Method <Action Method Name> ([Variable]) =
          <Action>
    when <Exp>

Rule ::=
    rule <Rule Name>:  when <Exp> =>
          <Action>

Exp ::=
    <Constant>
  | <Variable>
  | <Read Method Call>
  | <Exp> <Primitive Op> <Exp>
  | <Exp> ? <Exp> : <Exp>
  | (<Exp>) when <Exp>
  | <Local Binding> <Exp>

Primitive Op ::= + | - | & | …

Read Method Call ::=
    <Module Instance Name> . <Read Method Name> ([Exp])

Action ::=
    [Action]
  | <Action Method Call>
  | if <Exp> then <Action> else <Action>
  | <Action> when <Exp>
  | <Local Binding> <Action>

Action Method Call ::=
  <Module Instance Name> . <Action Method Name> ([Exp])
```

Figure 3-1: MRL grammar

Since the MRL grammar refers to an abstract syntax, it does not explicitly specify the syntax to delineate groupings of actions, local bindings, etc. However, we assume that such groupings are implied by the use of parentheses, braces, etc. in sample programs.

With regards to naming, the MRL language does not place restrictions on how design elements (state elements, interfaces, etc.) can be named. However, we usually adhere to the guidelines in Figure 3-2 when naming program components, especially when talking abstractly about a program property rather than about a concrete example.

Module Instance Name	::=	m_1 | m_2 | … | top
Module Definition Name	::=	mkFIFO | mkALU | mkGCD | …
Primitive Module Name	::=	mkReg
Primitive Instance Name	::=	r_1 | r_2 | ... // registers
Read Method Name	::=	f_1 | f_2 | …
Action Method Name	::=	g_1 | g_2 | …
Read or Action Method	::=	h_1 | h_2 | …
Variable Name	::=	t_1 | t_2 | …
Rule Name	::=	R_1 | R_2 | …

Figure 3-2: MRL naming conventions

3.1.2 Rules

As the name implies, rules are the key concept behind rule-based descriptions. The structure of a rule in MRL programs is identical to that used by Hoe and Arvind in their synthesis framework: it is an atomic action (body) that is protected by a guard. We will also call a rule's guard its *when* condition or predicate. The key difference between rules in an MRL program and rules in Hoe and Arvind's framework is that the rule guard and rule body can now make calls to the interface methods of arbitrary modules, not just primitive modules. As we will see, compiling rules that make calls to user-defined methods poses new challenges when generating efficient schedulers for the design.

3.1.3 Interface methods

User-defined interface methods are the key difference between modular rule-based (MRL) programs and the rule-based flat programs that Hoe and Arvind considered. Interface methods are the mechanism that allows rules and methods in different modules to communicate with

one another. As we will see in Chapter 4, scheduling and constraining the use of interface methods so that atomicity of rule execution is ensured is an interesting and important problem.

We distinguish between two types of interface methods: *read* methods and *action* methods. *Read* methods return a value (for example, the FIFO *first* method) and do not update a module's internal state. *Action* methods update a module's state and do not return a value (for example, the FIFO *enq* method). Since read methods return values, they are called from within expressions (*Exp*). Action methods update state and hence are called inside a rule or inside another action method's body, but not from within read methods. We often refer to action method calls simply as *actions*.

A very innovative feature that we have adopted from Bluespec, and which is not found in other languages, is a method's *implicit condition*. This condition determines whether or not a method is allowed to be called. For example, the implicit condition of the FIFO *enq* method is true only if the FIFO is not full. If the implicit condition is false (the FIFO is full), then the method (*enq*) must not be invoked.

Since rules must execute atomically, either all its actions or none of them must execute. Hence, if one of the rule's actions has an implicit condition that is false, the rule cannot execute. (We relax this restriction slightly in a later section when we consider an action block that contains calls to action methods within an *if* statement.)

The syntactic structure of interface methods in MRL programs is very similar to the rule syntax: each method contains a *when* condition (the implicit condition) and a body which either performs a set of actions if it is an action method or returns a value if it is a read method. One difference between methods and rules is that methods can accept input parameters.

3.1.4 Actions

Actions define the state update function of rules and action methods. The simplest action is a register write: *x := y*. More complex actions can make calls to user-defined action methods, for example a FIFO enqueue: *f.enq(x)*. This notation says to call module instance *f*'s *enq* method with input parameter *x*.

Actions can also consists of multiple method calls, for example: *x := y; y := x;* is an action that contains two register writes and two register reads. We interpret groupings of actions inside rules and methods the same way that Hoe and Arvind did: when multiple actions

appear within a rule or method, they must execute in parallel. This means that all state must be read before any updates occur. In the above example this means that the values of *x* and *y* should be swapped, rather than sequentially assigned. Because of their parallel interpretation, we can arbitrarily reorder actions within a sequence of actions. Another implication of this parallel interpretation is that two actions within a sequence must not write to the same state since the outcome would not be well defined. We mark any program in which multiple updates to the same state occur within a sequence of actions as invalid.

We allow two conditional constructs to appear within actions: *if* statements and *when* clauses. We will examine the execution semantics of these two constructs in more detail in Section 3.2.2. However, at a high level, these constructs conditionally prevent actions from executing. The key distinction is that if a *when* condition evaluates to false then the entire rule or action method must not be executed. In contrast, if an *if* statement's predicate is false, then the *if* statement's body must not be executed, but this does not disable the entire rule from executing.

3.1.5 Local bindings

Local bindings allow an expression to be assigned to a variable. Use of the variable name inside another expression has the equivalent meaning of textually substituting the expression that is bound to the variable. Hence, this construct is really a programming convenience but we include it in our language because it will be helpful throughout the thesis when we write programs and program transformations.

Local bindings can appear within modules, rules, methods, actions, etc. We assume that conventional scoping rules apply to the variable names.

3.1.6 Module hierarchy

The MRL language syntax allows any module to call any other module's interface methods. However, we place restrictions on what types of module interactions are valid. The synthesis algorithms in the following chapters also only apply to a subset of the valid module structures. To better understand these restrictions we introduce two graph structures.

The restriction for a module hierarchy to be valid is that its method calls must not be mutually recursive. This means if we construct a *method call graph* as follows, the graph must be acyclic for the MRL program to be valid:

- Each interface method corresponds to a node in the graph.
- We draw an edge from node $m_a.h_1$ to $m_b.h_2$ if method $m_a.h_1$ makes a call to method $m_b.h_2$.

The rationale for requiring this acyclic method call structure is that we must be able to statically generate hardware for a MRL program. If methods were to make mutually recursive calls (their method call graph forms a cycle), then dynamic elaboration would be required to determine when the recursive calls end. None of our synthesis algorithms fit into a framework in which a single method's execution takes an indeterminate amount of time and resources. Hence, we mark such MRL programs as invalid.

We introduce *module call graphs* to understand which MRL programs we can efficiently synthesize. They are defined as follows:

- Each module instance in the MRL program corresponds to a node in the module call graph.
- We draw an edge from node m_i to m_j if and only if module m_i makes a call to an interface method of module m_j.

In general, our synthesis algorithms only apply to module call graphs that form a tree. This means that each module instance's interface methods can be called from one module only. Many designs satisfy this restriction and those that do not can be transformed such that their new module call graph does form a tree. Hence, all valid MRL programs will be synthesizable. Unless we indicate otherwise, it should be assumed that the algorithms we present in this thesis only apply to module call graphs that form trees.

Since at some point we have to instantiate real hardware (for example registers), all designs must contain primitive state elements at the leaves of their call graphs. As in Hoe and Arvind's research, we assume that primitive elements are well understood and that we know how to generate the logic that interfaces to them and that schedules them. Most of this thesis assumes that there is only one primitive state element: registers in the early chapters, and a derived element, the EHR in later chapters. We will show how most other elements, including elements that were previously considered primitive, such as a FIFO, can be built from the primitive register without sacrificing performance.

3.1.7 Syntactic sugar

The MRL language contains several syntactic constructs which are not required to explain the semantics of a modular rule-based language. However, we include these constructs because they will be convenient to use in later chapters. One such construct is "*if then else"*. This can be desugared as shown below. We are simply splitting the *then* and *else* part of an *if* statement. For readability we will continue to use *"if then else"* in our examples, but only include *"if then"* statements in our language transformations:

```
if <Exp> then                          if <Exp> then
  Actions asT                            Actions asT;
else                          ⇔        if (!<Exp>) then
  Actions asF                            Actions asF
```

Since register reads and writes occur frequently in example programs we allow for special abbreviated syntax for register access: a register name in an expression implies a call to its read method. The *:=* operator is equivalent to invoking the left-hand-side's write method with the right hand side's expression as its input argument. Hence, the following translations can be applied at all times (in either direction):

```
r := e              ⇔       r.write(e)
The Exp: r          ⇔       r.read()
```

Finally, it turns out that implicit conditions are also a form of syntactic sugar. We will explain this in detail in Section 3.2.2 (When lifting). However, the pairing of methods with conditions is such a convenient construct that we will continue to use it in our semantic discussion.

3.1.8 MRL vs. Bluespec and ATS

The reader will note that the modular rule-based language Bluespec[4] is a much richer language than MRL. The key difference is that Bluespec contains a sophisticated type system and an advanced pre-processor. MRL can be thought of as an intermediate language of Bluespec that contains all structural and interesting rule properties, but from which types have

been stripped away and in which the pre-processor has inlined functions, substituted compile-time parameters, etc. The contribution of this chapter is not the language itself, but rather in articulating the execution semantics that such a modular rule-based language must adhere to.

In the context of modules, Bluespec had introduced modules in an object-oriented framework before we began this research. However it took some time to fully understand what the semantics of such a modular rule-based language should be. Fully understanding and specifying the semantics allowed us to create an efficient modular compilation flow which we present in the next chapter.

An example of a powerful and important Bluespec feature that we do not include in MRL is the ability to parameterize modules. Bluespec modules can be passed values, logic, or even other modules when they are instantiated. This allows for sophisticated libraries to be created. For example, a FIFO can be parameterized on the number of elements it contains, or what function should be applied to each element of the FIFO when performing a search on its elements. Our language does not include module parameters. Although very powerful and useful when designing hardware, they do not change the semantics or compilation of a modular rule-based description. We can treat all of these constructs as strictly a pre-processing step which results in a MRL description.

We can also contrast MRL with the framework in which Hoe and Arvind worked. The key difference is that Hoe and Arvind allowed rules to only interact with primitive state-elements (registers, register files, and FIFO's). In contrast, MRL allows a designer to create new modules and enables rules to interact with the modules through interface methods. Such modular design is critical for large-scale hardware design and as we will see poses some interesting challenges.

3.2 MRL to FRL translation

Now that we have a basic understand of the MRL language we can focus on the technical contribution of this chapter, which is to explain the precise meaning of a MRL program. We specify its meaning by providing a syntactic translation from MRL to FRL, where FRL is a flat rule-based language, equivalent to Hoe and Arvind's ATS framework. For completeness we also present a formal interpretation of FRL programs. The motivation for this two-step process rather than a direct interpretation of MRL is that we already understand FRL as a base-line

model for guarded atomic actions. From Hoe and Arvind's research we also understand how to generate hardware from FRL style algorithms. Thus, we think the reference model for a modular rule-based description (MRL programs) is best described via a flattened design (FRL programs).

```
Program ::=
    [Primitive Module Instance]
    <Module Definition>

Module Definition ::=
    module <Module Definition Name>
          [Local Binding]
          [Rule]

Primitive Module Instance ::=
    <Primitive Module Name> <Primitive Instance Name>

Local Binding ::=
    <Variable> = <Exp>

Rule::=
    rule <Rule Name>:  when <Exp> =>
                    <Action>

Exp ::=
    <Constant>
  | <Read Method Call>
  | <Exp> <Primitive Op> <Exp>
  | <Exp> ? <Exp> : <Exp>
  | [Local Binding] <Exp>

Primitive Op ::=
    + | - | & | …

Read Method Call ::=
    <Primitive Instance Name> . <Read Method Name> ([Exp])

Action ::=
    [Action]
  | <Action Method Call>
  | if <Exp> then <Action>
  | <Local Binding> <Action>

Action Method Call ::=
    <Primitive Module Name> . <Action Method Name> ([Exp])
```

Figure 3-3: FRL grammar

A grammar for the FRL language is provided in Figure 3-3. It is a subset of MRL where the key difference is that FRL programs do not contain a module hierarchy. All MRL module instances must be primitive state elements, and hence all method calls can be to primitive state elements only. (Using Hoe and Arvind's framework, we can assume that it is understood how to compile a set of rules that interact with primitive elements only.) To make FRL equivalent to Hoe and Arvind's synthesis language, we also require that *when* clauses only appear in the predicate of each rule, not in the rule body as was possible in MRL.

The translation of a MRL program into FRL occurs in two steps: (i) flatten the design through repeated merging of module instances until only a single top level module remains and all method calls are to primitive state elements, and (ii) lift conditional *when* clauses to enforce the atomic rule property (the *when* clauses appear during the merging process). We describe both of these steps in the next subsections.

3.2.1 Flattening

They key to the translation of MRL into FRL is the removal of the module hierarchy. We accomplish this flattening process via repeated merges of MRL modules until only a single top-level module remains. This top level module by definition will make calls to primitive state elements only.

The MODMERGE procedure in Figure 3-4 merges two arbitrary (non-primitive) MRL module instances m_1 and m_2. The procedure produces a new module m_{merged} which behaves the same as the two original modules m_1 and m_2. Merging takes place in four steps. First we create m_{merged} by adding all state, rules, local bindings, and methods of m_1 and m_2 in the new module. Since m_1 and m_2 will be removed after merging we then have to remove all references to their methods. References from modules other than m_{merged} can simply be redirected to call the corresponding method of m_{merged} rather than m_1 or m_2. However, if m_{merged} makes a call to either m_1 or m_2's methods, then we must inline the corresponding method since we do not permit a module to call its own methods. As shown in Figure 3-5 we use a conventional interpretation of inlining: we bind the method parameters with the values used in the method call and inline the entire method body, including the implicit condition (as a *when* clause). The final step in the merging process is to remove the original modules m_1 and m_2.

We assume that there are no naming conflicts between m_1's and m_2's methods (if there were, we would have to add a renaming step). As described in Section 3.1.6 we also assume that interface methods do not make mutually recursive calls as the inlining procedure would otherwise not be well-defined.

Now that we understand how to merge two modules, we can flatten an entire design through repeated merging of modules. This process is shown in Figure 3-6.

```
ModMerge(m1, m2) =
 1. Define a new module m_merged such that     // union of:
      m_merged.state   = m1.state ∪ m2.state;  // local instances
      m_merged.rules   = m1.rules ∪ m2.rules;  // rules
      m_merged.lb      = m1.lb ∪ m2.lb;       // local bindings
      m_merged.meth    = m1.meth ∪ m2.meth;   // interfaces

 2. Substitute module name m_merged for all uses of module
    names m1 and m2  in other modules

 3. foreach method call mi.h in m_merged where mi ∈ {m1, m2}
      inline the method call mi.h

 4. Remove modules m1 and m2
```

Figure 3-4: The MODMERGE procedure

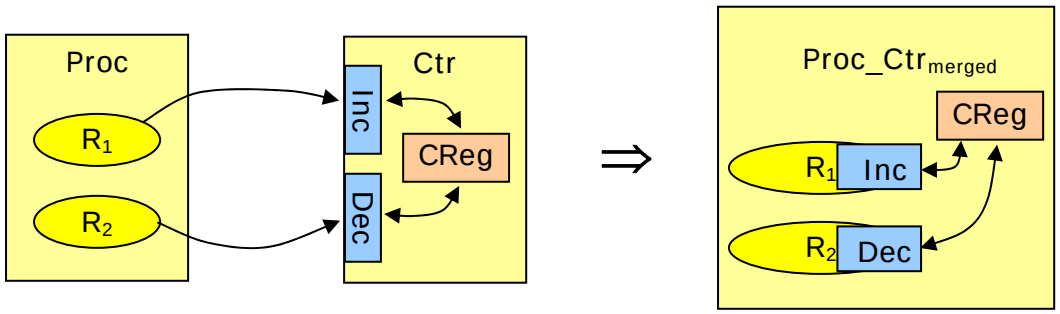
```
Suppose we are given read method m.f(x) and an action method
m.g(x):
   m.f(x) = e_f when (e_p);
   m.g(x) = a  when (e_p);

To inline these methods means to replace calls (m.f(e_x) and
m.g(e_x)) as follows:
   m.f(e_x) ≡ e_f[e_x / x] when (e_p[e_x / x]);
   m.g(e_x) ≡ a[e_x / x]  when (e_p[e_x / x]);

An alternate inlining:
   m.f(e_x) ≡ x = e_x; e_f when e_p;
   m.g(e_x) ≡ x = e_x; a when e_p;
```

Figure 3-5: Inlining

```
FLATTEN =
   1.  while (the design contains more than one module)
         a.  pick two module instances m1 and m2
         b.  ModMerge(m1, m2)
```

Figure 3-6: The FLATTEN procedure

To better understand module merging consider the example in Figure 3-7. In this example we show how two modules (Proc and Ctr) are merged into a single module. During the merging process the CReg register is inserted into the merged module, the methods (with parameters) are inlined into the rules R_1 and R_2, and the original modules Proc and Ctr are removed. In the next section we discuss what it means to have *when* clauses inside a rule (as the new R_1 and R_2 rules have). We will also show how the *when* clauses can be lifted to the rule predicate. (Note: p_1, p_2, and p_3 are placeholders for boolean expressions. a_1 and a_2 are placeholders for actions.)




```
module Proc
   Ctr c;

   rule R1: when (p1) =>
     a1;
     c.Inc(1);

   rule R2: when (p2) =>
     a2;
     if (p3) then c.Dec(2);
endmodule
```

```
module Ctr
   reg CReg;

   method Inc(x) =
      CReg := CReg + x;
   when (CReg < 127);

   method Dec(x) =
      CReg := CReg - x;
   when (CReg > 0);
endmodule
```

```
module Proc_Ctr merged
   reg CReg;

   rule R1: when (p1) =>
      a1;
      ((CReg := CReg + 1) when (CReg < 127));

   rule R2: when (p2) =>
      a2;
      if (p3) then ((CReg := CReg - 2) when (CReg > 0));
endmodule
```

Figure 3-7: Proc / Ctr module merge

A couple important properties of the FLATTEN procedure are worth pointing out: the procedure terminates for all *valid* programs and the procedure produces a unique top-level result, regardless of the order that MODMERGE is applied to the modules in a design. Termination is obvious since each call to MODMERGE inside FLATTEN adds one module and removes two—reducing the total number of modules by one. Hence, eventually we must be left with just a single top level module and all method calls in this top level module must be to primitive state elements. If there was a call to a non-primitive module then we could apply the MODMERGE procedure again. Notice, all steps except for inlining within step 3 of the MODMERGE procedure take finite time. Step 3 terminates as long as the method calls of m_1 and m_2 do not form a cycle.

3.2.2 When lifting

After a design has been flattened via the FLATTEN procedure we are left with a single top-level module which nearly satisfies the FRL grammar. To make the module a FRL description we need to lift the *when* clauses that appear in rule bodies up to their corresponding predicates. These *when* clauses appear inside expressions and actions of rule bodies due to the implicit conditions that were inlined during the FLATTEN procedure. An example where this happens was shown in Figure 3-7.

The reader should recall that the intent of implicit conditions is to prevent a method from being called if the condition is false. In addition, so as to ensure atomicity, if one of a rule's methods cannot be called because its implicit condition is false, then none of the rule's methods should be called.

One procedure to lift *when* clauses is to remove all *when*'s from the rule body and conjugate them with the rule predicate. This would satisfy the condition that the rule does not execute unless all the implicit conditions of the methods that were called are true. However, it is more constraining than required. This becomes clear if, as shown in Figure 3-8, we lift the *when* clauses from the Proc / Ctr example in this manner. By lifting the "(CReg > 0)" expression (which originated from the Dec implicit condition) into R_2's predicate, we prevent the rule from executing whenever CReg is 0. However, the Dec method of the original Ctr module would only have been invoked if p_3 was true. Thus, if p_3 is false, it is alright to execute rule R_2 regardless of what the state of CReg is. We show the new R_2 rule that implements this

style of *when* lifting in Figure 3-9. (Note: rule R_1 is the same regardless of whether the simple or conditional when lifting procedure is applied since it does not contain conditionals.)

```
module Proc_Ctr_merged
   reg CReg;

   rule R1: when (p1 & (CReg < 127)) =>
      a1;
      (CReg := CReg + 1);

   rule R2: when (p2 & (CReg > 0)) =>
      a2;
      if (p3) then (CReg := CReg - 2);
endmodule
```

Figure 3-8: Simple when lifting

```
   rule R2: when (p2 & ((!p3) | (CReg > 0)) =>
      a2;
      if (p3) then (CReg := CReg - 2);
```

Figure 3-9: Conditional when lifting

It is important to recognize that we made a choice in how *when* conditions should be lifted. Either approach works and implies slightly different semantics since the allowable behaviors are different in the two cases. Although slightly more complex, we choose the second approach because in some cases it leads to better performance by allowing rules to execute when method calls whose implicit conditions are false are located inside *if* statements whose condition is also false.

For completeness, we show a full *when* lifting procedure in Figure 3-10. We write this procedure as a source to source transformation. Any code that matches a description on the left hand side of these rewrite rules should be transformed into the corresponding code on the right hand side of the transformation. In these transformations e_i refers to expressions, a_i refers to actions, p_i refers to expressions in a *when* clause, and R is a rule. The previously described case lifting of when's across conditionals is marked as "SPECIAL CASE" in the procedure.

After no more of the *when* lifting transformations can be applied we are left with rules that contain only a single *when* clause—the rule predicate. At this point we have transformed a

modular rule-based description (MRL programs) into a corresponding flat description that can be simulated or synthesized using the basic model of guarded atomic action.

```
/*** when's are lifted across expressions ***/
(e1 when p1) when p2              =>  e1 when (p1 ∧ p2)
(e1 when p) <Primitive Op> e2     =>  (e1 <Primitive Op> e2) when p
e1 <Primitive Op> (e2 when p)     =>  (e1 <Primitive Op> e2) when p
(e1 when p) ? e2 : e3             =>  (e1 ? e2 : e3) when p

/*** Conditionals of parameters are lifted.   These   ***/
/*** method calls must be to primitive modules  since ***/
/*** flattening / inlining occurs before when lifting ***/
a(…, e when p, …)                 =>  a(…, e, …) when p
e1(…, e2 when p, …)               =>  e1(…, e2, …) when p

/*** Conditionals are lifted across actions ***/
if (e when p) then a              =>  (if e then a) when p
a1; (a2 when p)                   =>  (a1; a2) when p
(a when p1) when p2               =>  a when (p1 ∧ p2)

/*** SPECIAL CASE ***/
if e then (a when p)              =>  (if e then a) when (p ∨ ~e)
e1 ? (e2 when p) : e3             =>  (e1 ? e2 : e3) when (p ∨ ~e1)
e1 ? e2 :(e3 when p)              =>  (e1 ? e2 : e3) when (p ∨ e1)

/*** lifting conditionals to the rule predicate ***/
rule r: when p1 => (a when p2)    => rule r: when (p1 ∧ p2) => a

/*** remove when conditions from temporary bindings ***/
t = et when p; e                  =>  t = et; e[(t when p) / t]
t = et when p; a                  =>  t = et; a[(t when p) / t]
t = et when p; R                  =>  t = et; R[(t when p) / t]
```

Figure 3-10: When lifting transformations

As an aside, it should now be clear that implicit conditions turn out to be syntactic sugar. We could split each method into two methods: its body and a read method corresponding to its implicit condition. For example, if we have method *m.h*, we could split it into *m.h_body* and *m.h_cond*. *m.h_body* performs the action when called (or returns a value if *m.h* is a read method). *m.h_cond* is a read method that returns the value that the implicit condition of *m.h* would have computed. (Note: neither *m.h_body* nor *m.h_cond* has an implicit condition.) We can then replace all calls to *m.h* with *"m.h_body* ***when*** *m.h_cond"*. By the above definitions, this has precisely the same meaning.

3.3 FRL execution semantics

We now present a precise meaning of FRL programs. This section should be read as an aside and the details are not important to understanding the remainder of the thesis since the high-level idea of sequential and atomic execution of rules applies to FRL programs. However, we have developed a simple operational evaluation function for FRL programs and present it here to complete the picture of what it means for an MRL program to execute.

In our execution model we utilize tables to maintain state (S), state updates (U), and temporary values (T). For completeness, we present a definition of a table below. Here, A and B are place holders for any of the tables S, T, and U. These tables can be thought of as lists of assignments (pairs <r, v>) in which later assignments (the right hand side of a "+") to an entry *r* take precedence over earlier assignments (the left hand side of the "+").

```
Table definition:

A[r] = v if <r, v> ∈ A
       ⊥ otherwise

(A + B)[r] = v_b if <r, v_b> ∈ B
             v_a if <r, *> ∉ B and <r, v_a> ∈ A
             ⊥ otherwise
```

We show an interpretation of FRL programs in two steps. First we explain what it means for a rule to execute. We then show what it means for rules to execute in sequence.

3.3.1 Rule execution

Each FRL program can be divided into two sections: a local bindings section (*LB*), and a rules section. Each binding in *LB* takes the form: *t = Exp;* and each rule takes the form: *rule* R_i*: when* (π_i) => a_{Ri}. Hence, we refer to rule R_i's predicate by π_i and its action by a_{Ri}. Given these definitions we can write the following program to define the meaning of "execute rule R_i". This program evaluates the local bindings, not all of which the rule has to use, and then executes the rule's actions (a_{Ri}), provided the predicate (π_i) is true:

```
LB;
if π_i then a_i;
```

The exact meaning of this program can be explained via an operational evaluation function (E_{Rule}). This function takes as input the above program and the current state of the system (S). It returns the new state of the system after the program has executed. E_{Rule} can be best explained in two steps: (i) it computes a list of state updates (U), and (ii) it applies those updates to the states ($S + U$). The motivation for splitting the evaluation into two phases is that we interpret *read*'s to happen before *write*'s take effect, even if the read appears later in the program. By accumulating all state updates (*write*'s) before applying them, we can ensure that reads do not observe effects they should not see during sequential execution.

E_{Rule} computes the list of state updates using another operational evaluation function (E_a). E_a accepts as input a program, the current state (S), as well as a table of temporary assignments (T)—initially empty. The intent of the table of temporary assignments is that it is valid only during a single execution. When another rule is evaluated, the temporary values are recomputed. In contrast, the state S must persist from one execution to another since it represents register state. Hence, the updated state will pass from one execution to the next.

```
E_Rule signature:  Program -> State table -> New state table

E_Rule ⟦LB; if π_i then a_i;⟧ S =
      let U = E_a ⟦LB; if π_i then a_i⟧ S ∅ in
        S + U
```

Below we present the definition of E_a. This definition is a sequential interpretation of the input program to the function. Two of the reductions in E_a stand out: "*t = e; a*" and "*r := e; a*". The first case assigns an expression (*e*) to a temporary (*t*) and then evaluates the action. This is expressed by adding (using the symbol "+") the pair consisting of *t* and the evaluation of *e* to the environment *T*. We then evaluate the action using the updated environment. The second case corresponds to a register assignment (*r := e*), followed by an action (*a*). As mentioned, we must not immediately update the state *S* to reflect the change in register value since a later action within the same rule should not observe the change. Thus, we add the assignment to the list of updates that must be performed when E_a finishes evaluating the entire program. (Note: this evaluation function requires that temporary variable definitions (local bindings) occur before their use. If an input program does not satisfy this condition, then it can be transformed to satisfy the condition by performing a topological sort on the variable uses / definitions—provided of course a definition for each variable that is used exists.)

```
Ea signature:
    Action -> State table -> Temps table -> State update table

Ea ⟦∅⟧ S T                        =  ∅
Ea ⟦t = et; a⟧ S T                =  Ea ⟦a⟧ S (T + <t,(Ee ⟦et⟧ S T)>)
Ea ⟦(if e then a1); a2⟧ S T       =  Ea ⟦(if (Ee ⟦e⟧ S T) then a1);
                                        a2⟧ S T
Ea ⟦(if true then a1); a2⟧  S T =  Ea ⟦a1; a2⟧ S T
Ea ⟦(if false then a1); a2⟧ S T =  Ea ⟦a2⟧ S T
Ea ⟦r := e; a⟧ S T                =  <r, (Ee ⟦e⟧ S T)> + (Ea ⟦a⟧ S T)

Note: any of the actions in the above functions can be empty.

an additional rule propagates undefined values (⊥):

Ea ⟦(if ⊥ then a1); a2⟧ S T       =  ⊥
```

Next, we define the evaluation function for expressions, E_e. It takes as input an expression, the current state *S*, and the table of temporary assignments *T*. It returns the value of the expression (or ⊥ if an error occurrs).

```
Ee signature:
    Expression -> State table -> Temps table -> value

Ee ⟦c⟧ S T                  =  c
Ee ⟦r.read()⟧ S T           =  S[r]
Ee ⟦t⟧ S T                  =  T[t]
Ee ⟦e1 op e2⟧ S T           =  op((Ee ⟦e1⟧ S T), (Ee ⟦e2⟧ S T))
Ee ⟦e1 ? e2 : e3⟧ S T       =  Ee ⟦(Ee ⟦e1⟧ S T) ? e2 : e3⟧ S T
Ee ⟦true ? e2 : e3⟧ S T     =  Ee ⟦e2⟧ S T
Ee ⟦false ? e2 : e3⟧ S T    =  Ee ⟦e3⟧ S T
Ee ⟦t = et; e⟧ S T          =  Ee ⟦e⟧ S (T + <t,(Ee ⟦et⟧ S T)>)

Note:  S[r] always returns a value since register values
persist.

T[t] returns ⊥ if t has not been bound inside T—this is an
error condition.

Additional rules to propagate ⊥ are listed below:

Ee ⟦e1 op ⊥⟧ S T            =  ⊥
Ee ⟦⊥ op e2⟧ S T            =  ⊥
Ee ⟦⊥ ? e2 : e3⟧ S T        =  ⊥
```

Thus, we have presented a precise description of what it means to execute a rule. Next we can define what it means to execute rules in sequence.

3.3.2 Sequential execution of rules

The key property of rule execution is that rules must appear to execute in sequence. We can now specify what it means to execute rules in sequence. Suppose we are given an initial state *S*, local bindings *LB*, and rules R_1 and R_2. Sequential execution of these rules, denoted by R_1 \$ R_2 is defined as follows. It takes the initial state of the system (S) and returns the next state.

```
R1 $ R2 ≡ let S' = E_Rule ⟦LB; R1⟧ S in
            E_Rule ⟦LB; R3⟧ S'
```

Given a system of *n* rules (R_1, …, R_n) we can then construct a program that repeatedly performs round robin scheduling of the rules:

```
while (true) do
   S := R1 $ … $ Rn;
```

We can also describe a scheduler which in each iteration selects *one* rule whose predicate is true and then executes that rule. This closely resembles the baseline circuit that we describe in Chapter 2:

```
while (true) do
   // compute rule predicates
   tπ1 := Ee ⟦LB; π1⟧ ;
   …
   tπn := Ee ⟦LB; πn⟧ ;

   // use a scheduler to compute pi's
   {ϕ1, …, ϕn} := BaselineSchedule(π1, …, πn)

   // execute the rule whose ϕ is true
   if (ϕ1) then S := E_Rule ⟦LB; R1⟧ S
   …
   if (ϕn) then S := E_Rule ⟦LB; R1⟧ S
```

3.4 Chapter summary

The key contributions in this chapter are the introduction of a modular rule-based language (MRL) and a specification for how such a language should behave. We explain its behavior via a flattening procedure which eliminates the module hierarchy. This results in a set of rules which after *when* lifting are equivalent to the conventional framework of guarded atomic actions. Finally, we presented an operational evaluation function that shows precisely what it means for rules to execute in sequences. In summary, we have defined how modular rule-based descriptions should execute. This will serve as the reference model when we perform true modular compilation in the next chapter.

Chapter 4

Modular Compilation

A modular synthesis flow is essential for a scalable and hierarchical design methodology. Modularity is important because it enables the exchange of reusable IP, because it facilitates verification, and because modular compilation can significantly improve synthesis times. The previous chapter presented a modular language for guarded atomic actions (MRL). However, we have only shown how this language can be translated into a flat language (FRL), which in turn could be synthesized into hardware using Hoe and Arvind's synthesis algorithms. This chapter presents a true modular compilation algorithm that generates circuits without first flattening the design[47]. An important requirement is that the modular circuits will continue to match the semantics of the flattened modular description. The key contributions are (i) we introduce a set of *scheduling annotations* for module interface methods that constrain their use; (ii) we show how rules and interface methods can be scheduled given the scheduling constraints of the methods they call; (iii) we show how a module's scheduling annotations can be derived, provided that the scheduling annotations for all modules the module communicates with are known; and (iv) we show how to generate the glue logic that connects modules to one another. As we will see, it is not possible to guarantee atomic execution of rules without these scheduling annotations.

Although the constraints pose a challenge during synthesis, we show that they significantly improve on the traditional style of informal module interface specification and

use. Because the scheduling annotations represent formal specifications on how interfaces can be used, they facilitate the exchange of reusable IP, encourage a "correct by construction" design methodology, and allow for easier architectural exploration by allowing modules to be swapped in and out of a design without requiring the connecting modules to change. We view these aspects as crucial components to improve the hardware design process.

Even though we now view the above aspects as the most important contribution of the modular compilation flow, the initial motivation for introducing modular compilation grew out of a more immediate practical need that arose as we were designing a processor using an early version of Bluespec. We had written a highly-parameterized FIFO with a recursive search function that was used to generate values for the processor bypass network. This description was more parameterized than would be required for a specific processor implementation but our expectation still was that with proper synthesis algorithms the final circuit would be equivalent to a hand-coded RTL implementation. If successful, this highly parameterized FIFO could be used across many different designs. Unfortunately, after flattening, synthesis time for the processor was excessive and scheduling results were unsatisfactory—only alternating processor stages executed concurrently because the FIFO did not permit simultaneous enqueue and dequeue operations. Both of these issues are addressed in the modular flow because modules can now be compiled separately (thereby dramatically improving the compile times), and we allow for user-prescribed interface scheduling which the user can take advantage of when he has some high-level knowledge that the compiler is not able to derive. Chapter 4 improves on this idea of user-prescribed interface scheduling by allowing the designer to specify arbitrary performance constraints on module interfaces without risking that the underlying semantics are altered.

To frame the context of this work, we should note that Bluespec was an object oriented and modular language before we began this research. The language had the power to express FIFO's, arrays and many other hardware building-blocks as user defined modules using only registers. However, similar to the process described in the previous chapter, the compilation flow flattened the design until only rules that interacted with primitive elements remained. Hoe and Arvind's[27-29] analysis and synthesis algorithms were then applied to generate RTL Verilog. However, as mentioned, this flow led to excessive compile times for larger designs, suffered from scheduling (throughput) problems, and did not present an abstraction for the reuse of precompiled IP. Thus, we take advantage of the language features developed in Bluespec, but the synthesis algorithms and interface constraints are the result of our work.

The next section presents an example that illustrates the challenges of modular rule-based compilation. We then introduce the module interface annotations and describe the modular compilation algorithm. The end of the chapter presents results and discusses possible improvements to the modular synthesis flow.

4.1 The goal of modular compilation

This chapter considers a modular flow in which each module has interface methods and the internal behavior of the module is described in terms of a set of guarded atomic actions on the state elements of the module. A module can also read and update the state of other modules, but only by invoking the interface methods of those modules. This is illustrated in Figure 4-1. The goal in modular compilation is to compile each of these modules individually while ensuring that the sequential and atomic execution of rules is maintained across module boundaries. In addition, a compilation algorithm that permits the maximal amount of concurrent rule execution is desirable.

For example, given the design in Figure 4-1, a modular flow would first compile modules "*2"* and "*3*" individually—generating a circuit description for each of these modules, along with some minimal information (what we call *scheduling annotations*) that allows other modules to connect to them. The annotations of modules "*2*" and "*3*" tell us how rules can be scheduled inside module "*1*" as well as how to generate the glue logic that connects module "*1*" to modules "*2*" and "*3*". In order to maintain a level of abstraction, the scheduling annotations will export only a small amount of information about the module's internals.

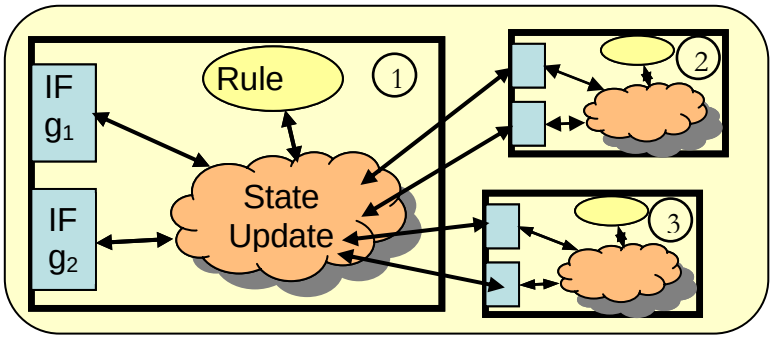


Figure 4-1: A modular design

As a concrete example we show the code for a two-element FIFO in Figure 4-2. The FIFO contains four registers—two to hold the data in each of the two FIFO elements, and two full registers to indicate whether the data registers hold valid data. We consider the "0" registers to be the part of the first FIFO element, and the "1" registers to be part of the second FIFO element. We always fill the first element before filling the second element. Hence, the FIFO will never have register full1 set to true while full0 is false. This FIFO contains the standard interface methods (enqueue—enq, dequeue—deq, clear, and first). In the next chapter we also show how to introduce bypasses.

```
module FIFO
  // local state definition
  mkReg data0;       // contents of FIFO element 0
  mkReg data1;       // contents of FIFO element 1
  mkReg full0;       // 1 if FIFO element 0 contains valid
                     // data, 0 otherwise
  mkReg full1;       // 1 if FIFO element 1 contains valid
                     // data, 0 otherwise

  // interface specification
  method enq(x) =
     data1 := x;     // can always write to data1
     full1 := full0;
     if (full0 == 0) then
      data0 := x;    // only write to data0 if FIFO was empty
     full0 := 1;     // contains at least one element after enq
  when (full1 == 0); // to enq, FIFO must no be full

  method deq =
     full1 := 0;
     full0 := full1;
     data0 := data1;
  when (full0 == 1); // to deq, FIFO must not be empty

  method clear =
     full1 := 0;
     full0 := 0;
  when (true);       // can be called anytime

  method first =
     return data0;   // return the first FIFO element
  when (full0 == 1); // FIFO must contain valid data

endmodule
```

Figure 4-2: 2-Element FIFO

We will use this code as a running example throughout the remainder of the thesis. Although it is a simple module, it contains many of the properties that make modular rule-based descriptions interesting, but also challenging.

In a modular compilation flow we compile the FIFO on its own, that is, separate from any other module. The result is a FIFO circuit description (RTL) along with a set of FIFO interface scheduling annotations. Rather than being produced by the designer, the RTL code and scheduling annotations could also have been provided as part of a reusable IP library that includes a precompiled FIFO module. The modular flow we present in this chapter answers two important questions about such a module: (i) how to connect to it, that is, what is the wiring protocol, and (ii) what are the constraints for using the methods. We outline these ideas for the FIFO module in the next paragraphs and provide a complete compilation flow in the following sections. The underlying semantic constraint in this work is that atomicity must be preserved across module boundaries—that is, the modular flow must exhibit behaviors that are permissible in the flat equivalent.

4.1.1 FIFO interface wiring

We expect the circuit that results from compiling the FIFO to have an interface as shown in Figure 4-3. The signals in this description have the following meanings, where *h* is a method name:

h_rdy: an output signal corresponding to the implicit condition of the method

h_en: an input signal that indicates the method should execute

h_data: the parameters that are passed / returned during the method call (note: this signal is usually a bus)

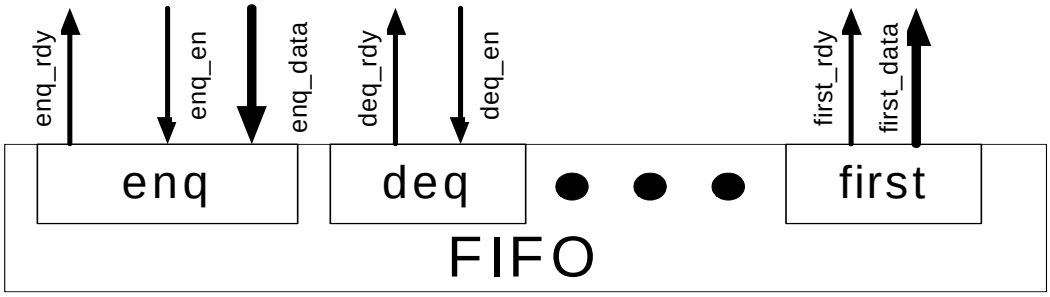


Figure 4-3: FIFO interface

Only action methods, that is, methods that update state have an *h_en* signal. Read methods do not require an enable signal because it is safe for read methods to always return a value, even if the result is not used. All modules we generate will have this style interface wiring since it incorporates a simple protocol that communicates the critical method call signals: data (input / output), enable (the method is being called) and ready (the method can be called).

Let us now examine how these signals would be used from an instantiating module. Figure 4-4 shows two rules that interact with the FIFO from within the module Top. Clearly, rule R_0 must only execute if the FIFO is not full. Thus, the *enq* method's implicit condition signal (*enq_rdy*) must become part of rule R_0's predicate. Also, whenever rule R_0 executes, the *enq_en* signal must be asserted and the value 5 must be passed on the *enq_data* bus. Similar connections must occur for rule R_1.

```
module Top

  mkFIFO f0;

  rule R0:  when (true) =>
    f0.enq(5);

  rule R1:  when (true) =>
    f0.deq();

endmodule
```

Figure 4-4: Simple use of FIFO module

4.1.2 FIFO interface scheduling

In the above example, we demonstrated the basics of connecting the two rules R_0 and R_1 to the FIFO f0. Assuming that the FIFO circuit is generated correctly, it is reasonable to assume that each of these two rules executes atomically and matches its flat equivalent if only one rule executes at a time. However, given that one of the synthesis goals is to maximize concurrent rule firings, we must ask: is it permissible to execute both of these rules simultaneously? The answer clearly depends on the FIFO implementation. As shown in Figure 4-5, the interaction of the two rules depends on the interaction of the enqueue and dequeue methods. Depending on how the methods read and update shared state, the resulting behavior may or may not be explainable as sequential execution of the two rules.

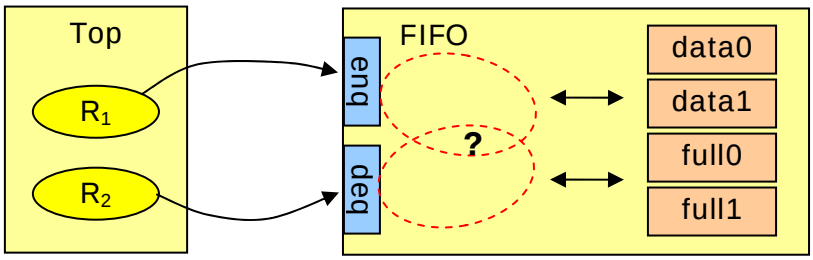


Figure 4-5: FIFO method overlap

If we assume that reads occur at the beginning of the cycle and writes occur at the end of the cycle, then 8 implementations are possible ($2^3 = 8$—since preferences for each write to data0, full0, and full1 can be given to either enqueue or dequeue). The implementations distinguish themselves based on which of the two methods takes precedence when both attempt to write to the same state. So long as only one rule executes at a time, it does not matter which implementation is chosen. However, the implementation choice can have important consequences when both rules simultaneously interact with the FIFO.

We can clarify this via an execution example. Given an initial FIFO state, depending on the FIFO implementation we could obtain the following results after enabling both rules (and corresponding wires connecting to the FIFO). (Note: "*x*" means don't care, and the values of *datai* do not matter if the corresponding *full* value is 0.)

	Circuit	full0	data0	full1	data1
Initial state		1	3	0	x
Outcome A	enq writes take precedence over deq writes	1	5	0	x
Outcome B	mixed precedence on writes to full and data	0	3	0	5

Outcome A arises if all writes by the enqueue method take precedence over the writes of the dequeue method. The resulting behavior can be explained as the execution of R_1 followed by R_0 (or also rule R_0 followed by R_1). However, outcome B is not consistent with the atomic execution of the two rules in either order, and hence is not a permissible execution. Such an outcome would arise if the FIFO circuit gives precedence of writes to the *full* registers to the *deq* method and gives the *enq* method precedence on writes to the *data* registers. (Note: by coincidence, outcome B is equivalent to executing R_1 followed by two executions of R_0 but

we were only considering the execution of two rules in this case.) Thus, depending on the FIFO implementation, rules R_0 and R_1 may or may not be able to execute simultaneously.

In a traditional RTL design flow it is the designer's responsibility to ensure that such module restrictions are observed. When using precompiled IP, or when interfacing to another designers block, the designer must read through manuals searching for this type of information. If a mistake is made, a hard to debug error will often arise. In the flow we introduce next, these interface properties are captured by scheduling annotations which specify whether methods can be simultaneously enabled. The designer may want to check the constraints to ensure proper performance, but this is not a functional correctness issue. Since the compiler schedules the rules that interact with the FIFO, the compiler will ensure that the rules do not execute concurrently unless the FIFO scheduling annotations indicate that such execution is consistent with atomic and sequential rule execution.

An additional benefit of this design style is that it allows for easy swapping in and out of modules. If a module that allows simultaneous execution of two of its methods results in a critical path that is too long we can replace it by an equivalent module that completely separates the two methods. The new module will have a smaller critical path but would not allow the two methods to execute together. Again, because these properties are captured by the scheduling annotations, the compiler would ensure that external rules are scheduled correctly.

In the next section we introduce the scheduling annotations. We then show how to schedule a set of rules that interact with methods whose annotations are known. After that, we show how rules should be connected to the modules they communicate with, and then show how to compile entire modules.

4.2 Interface method annotations

Scheduling annotations describe the pair-wise relationship of methods, say h_1 and h_2. Annotations must specify:

- if h_1 and h_2 can be called from a single rule
- if h_1 and h_2 are called from different rules can they be scheduled in parallel, and if so, then do they impose any ordering on those rules
- if h_1 can be called from two different rules simultaneously

The single rule specification is required because we have to ensure that the modular circuit matches its flat equivalent. When we enable both h_1 and h_2, the outcome must be the same as though the contents of the methods had been flattened into the module. Usually, this is simply a validity question, that is, does a valid flat meaning for the two methods exist? For example, suppose we are given the two action methods g_1 and g_2, a read method f_1, two rules R_1 and R_2, and registers r and x:

```
module m
  method g1:
    r := 1;
  when (true);

  method g2:
    r := 2;
  when (true);

  method f1:
    return r;
  when (true);
endmodule
```

```
rule R1: when (true) =>
    m.g1();
    m.g2();

rule R2: when (true) =>
    m.g1();
    x := m.f1();
```

If both methods g_1 and g_2 are enabled, then module m might give precedence to method g_2 and set r to 2. Hence, if module boundaries are maintained then the result of executing rule R_1 is that r is set to 2. However, if we flattened the design, we would obtain the rule:

```
rule R1: when (true) =>
   r := 1;
   r := 2;
```

This rule is not well defined because we are assigning to the same state twice, which violates the parallel semantics of the flat reference model. Thus, it is invalid to call g_1 and g_2 from the same rule, regardless of the modular implementation of g_1 and g_2. It should be clear that it is always invalid to call the same action method twice from within the same rule (provided the calls are not in mutually-exclusive branches of *if* statements). (Note: if it can be proven that two method calls produce the same result, then they can be called from the same rule. However, since we do not incorporate such a proof system in our compilation flow, we make the conservative assumption that two different method calls always produce different results.)

It turns out that this property is also important for read methods. In the next chapter we present compilation methods that allow read methods to observe values that action methods

write. For example, in the above example we expect that rule R_2 is valid since it has a well-defined flat meaning:

```
rule R2: when (true) =>
   r := 1;
   x := r;
```

If *r* is initially 0, then after executing R_2 we expect *r* to contain 1 and *x* to contain the original value of *r*: **0**. Suppose that module *m*'s circuit is such that f_1 returns the value of *r* **after** g_1 executes. Given a modular circuit in the above scenario, *r* would contain 1 and *x* would contain the new value of *r*: **1**, after executing rule r_2. Since this is different from the flat reference model, we would not be allowed to call f_1 and g_1 from within the same rule. In this case, the property of whether f_1 and g_1 can be called from the same rule depends on the implementation of the module *m*.

We use the symbol $h_1 \oplus h_2$ to indicate that flattening h_1 and h_2 into a rule has a well-defined meaning and that calling the two methods has the same meaning as the flattened version

Scheduling annotations must also address whether methods can be simultaneously called from multiple rules (or methods), and if so, whether there are any implied ordering constraints. This information will be crucial to determining which rules can execute simultaneously within each cycle.

We use the symbol $h_1 < h_2$ to indicate that if h_1 and h_2 are called simultaneously then the behavior is such that it appears as though h_1 executes followed by h_2. Using the notation from Chapter 3: $h_1 < h_2$ implies $h_1 ; h_2 \equiv h1 \$ h_2$

In the sample code above, depending on the module implementation, annotations "$g_1 < g_2$" or "$g_2 < g_1$" could be valid. For example, if the write to *r* by g_1 takes precedence over that of g_2, then "$g_2 < g_1$" applies. Similarly, if the circuits we generate do not forward values from one method to another, then we know that "$f_1 < g_1$" applies. However, if f_1 observes the value written by g_1, then "$g_1 < f_1$" applies. In general, "$f < g$" is true for all read methods *f* and all action methods *g* unless we generate circuits that forward values between methods.

Figure 4-6 shows the scheduling annotations that we use in the modular compilation flow. Each annotation combines the two types of properties that we discussed above: the

single rule behavior and the two rule behavior. During modular compilation we record these annotations for each pair of a module's methods in a *Conflict Matrix* (CM). An "X" in the table indicates that no valid behavior will be observed for that annotation in either the single or two rule case. Since a module does not know if two methods are being called from a single rule or from two rules, the single rule and two rule behaviors must clearly be equivalent if both are valid. (Note: in the example column of the table we assume a straight-forward module implementation in which all reads happen before writes, i.e. no value forwarding is allowed.)

Annotation	Single-Rule Behavior	2-Rule Behavior	Example
ME	don't care	don't care	h_1: e_1 when (x == 0) h_2: e_2 when (x == 1)
CF	$h_1 \oplus h_2$	$h_1 < h_2 \equiv h_2 < h_1$	h_1: x := 5 h_2: y := 6
<	$h_1 \oplus h_2$	$h_1 < h_2$	h_1: x := y h_2: y := 5
>	$h_1 \oplus h_2$	$h_2 < h_1$	h_1: x := 5 h_2: y := x
P	$h_1 \oplus h_2$	X	h_1: x := y h_2: y := x
$<_R, >_R$ / EXT	X	$h_1 < h_2 \neq h_2 < h_1$	h_1: x := 5 h_2: x := 6
$<_R$	X	$h_1 < h_2$	h_1: x := x+1 h_2: x := 6
$>_R$	X	$h_2 < h_1$	h_1: x := 6 h_2: x := x+1
C	X	X	h_1: x := x+1 h_2: x := x+1

Figure 4-6: Interface method annotations

Several things are worth noting about the annotations:

- The first case in this table is a special case: If two methods are *mutually-exclusive* (ME), that is, their guards (implicit conditions) can never simultaneously be true, then the methods obviously cannot affect each other since they will never be called

simultaneously. (A single rule that calls ME methods will never execute since the rule's guard will never be true.)

- The P annotation says that the parallel behavior of g_1 and g_2 ($g_1 \oplus g_2$) is not explainable as sequential behavior of g_1 and g_2. Hence, two rules containing such method calls cannot be scheduled simultaneously but it is permissible to call g_1 and g_2 within the same rule (or method).
- Even though annotation ($<_R$, $>_R$) makes sense we do not allow it for pragmatic reasons. It would require the scheduler to pass the information into the module about what order it has chosen for g_1 and g_2. We require the module to make this choice and specify it in its CM as $<_R$ or as $>_R$.
- Generally, an action method is not allowed to be invoked more than once from two different rules. However, there is one interesting exception which corresponds to the annotation EXT. Consider the action method "*g(a): x := a*". Suppose one rule calls "*g(3)*" and another rules calls "*g(4)*". It is possible to wire the module externally so that either argument 3 or 4 is passed to *g* and allow both rules to be scheduled concurrently. We indicate this property of an action method with the annotation EXT. EXT can only describe the relationship of an action method with itself. Hence, it will only appear as a diagonal entry in a conflict matrix.

4.2.1 Conflict matrices

As an example of a module's annotation, Figure 4-7 shows the CM (Conflict Matrix) for the primitive register element. The CM shows the pair-wise scheduling relationship between the read and write methods. For example, the "read < write" annotation specifies that the two methods can be simultaneously called from either one or two rules. The annotation also specifies that if simultaneously enabled, it will appear as though the read executes before the write executes.

$h_1 \setminus h_2$	read	write
read	CF	<
write	>	EXT

Figure 4-7: Register annotations

The next sections present algorithms that show how we schedule rules given the CM of the modules called. We will also show how the CM's in a module hierarchy can be derived, given only the register CM for the leaf nodes. However, this section first shows how important the specifications provided by the CM are. Using the FIFO example from earlier in this chapter we show how the "same" FIFO could have different implementations (CM's) that yield quite different behaviors. Usually these behaviors are captured as part of an English specification, which are easy to ignore, misinterpret, etc. In our modular compilation flow, these properties are captured in the FIFO's CM and are a core piece of each module—whether user prescribed or compiler-derived.

$h_1 \setminus h_2$	enq	deq	clear	first
enq	C	C	$<_R$	>
deq	C	C	$<_R$	>
clear	$>_R$	$>_R$	C	>
first	<	<	<	CF

(a)

$h_1 \setminus h_2$	enq	deq	clear	first
enq	C	$>_R$	$<_R$	>
deq	$<_R$	C	$<_R$	>
clear	$>_R$	$>_R$	C	>
first	<	<	<	CF

(b)

$h_1 \setminus h_2$	enq	deq	clear	first
enq	C	$<_R$	$<_R$	$<_R$
deq	$>_R$	C	$<_R$	>
clear	$>_R$	$>_R$	C	>
first	$>_R$	<	<	CF

(c)

Figure 4-8: Three FIFO conflict matrices

Figure 4-8 shows three possible conflict matrices (there are many more) for the FIFO module. Given the code in Figure 4-2, all three of these could be the result of modular synthesis. Similarly, a reusable IP library could make any one (or several) of these available to a designer. Since we assume that the compiler does not look at the internals of precompiled blocks, these CM's are the only information that is available about how the FIFO can be used.

Although all three CM's are mostly the same, the differences have a large impact in how their corresponding implementations can be used. The first CM (a), has "*C*" entries for the *enq* and *deq* method pairs. This means that the two methods must not be called simultaneously within the same cycle. As we saw earlier, it is easy to construct an implementation in which such a restriction applies since *enq* and *deq* might not act atomically on the FIFO state when simultaneously enabled. The second CM (b) allows *enq* and *deq* to execute simultaneously in different rules and it will appear as though *deq* executes followed by *enq* if both methods are enabled. In many cases, this is the desired FIFO behavior. The third CM (c) also allows *enq* and *deq* to execute simultaneously. However, in this case, it will appear as though *enq* happens followed by *deq*. This means that if the FIFO is initially empty then an enqueued value will fly through the FIFO if both methods are called (the value is enqueued and then immediately dequeued).

All three of these FIFO implementations are valid and are useful depending on the circumstances. Without a clear specification, such as the scheduling annotations inside the CM, it is hard for a user to understand what type of block is being worked with, and impossible for a compiler to deduce the necessary information. Thus, we believe these types of annotations must be part of a specification for hardware blocks to be easily reused. In addition, the scheduling algorithms that follow allow us to swap these different FIFO's in and out of a design without having to rewrite the rules that interact with them.

4.3 Module hierarchy

Most of our work on modular synthesis assumes a module instance call graph that forms a tree. This restriction simplifies the compilation process because it allows circuit generation and scheduling to occur on a module-by-module basis. In contrast, if we allow arbitrary call graphs, then one module's schedule can be affected by the actions of another module. Hence, either global knowledge about other modules is required or additional logic has to be added to

coordinate these modules' actions. We outline some of these strategies at the end of this chapter. However, we believe a tree-like call graph is reasonable for many designs.

We can always transform a design that does not satisfy the tree hierarchy condition into a design that does satisfy the condition. A naïve algorithm is to FLATTEN the entire design as was shown in Chapter 3. This results in a single top-level module that calls the methods of only primitive modules. (Primitive modules by definition do not interact directly with one-another.) However, we can do better than this by only merging modules that make calls to a shared module and those modules that are part of a cycling call structure. This algorithm is shown in Figure 4-9. An associated image that depicts its operation is shown in Figure 4-10.

```
MAKETREE =
 1.  while (the module instance call graph contains a cycle)
        a.  pick a set M of modules instances {m_a, m_b, …}
            that form a cycle
        b.  MODMERGE(M)

 2. while (a module exists whose methods are called by more
            than one other module)
        a. pick a set M of modules instances {m_a, m_b, …} that
            make method calls to a common module
        b.  MODMERGE(M)
```

Figure 4-9: MAKETREE algorithm

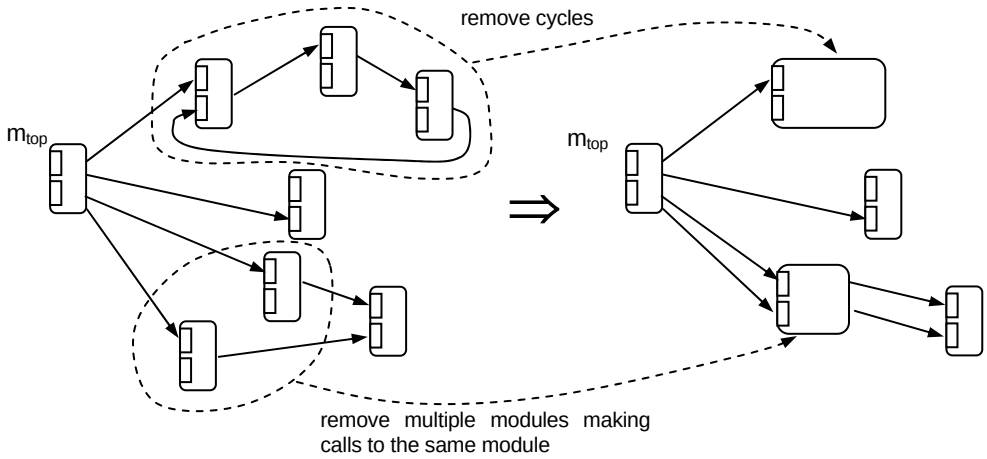


Figure 4-10: MAKETREE operation

(Note: The MAKETREE procedure assumes that MODMERGE can accept a set of modules as input. We previously defined MODMERGE to only operate on a pair of modules, but this can clearly be extended to a set of modules by repeated merging of module instance pairs. Unless otherwise specified, we assume throughout the remainder of this chapter that the module call structure has been transformed to be a tree—either by the designer or using the MAKETREE procedure.)

4.4 Rule scheduling using module interface annotations

The previous sections have discussed the benefits of a modular rule-based description and introduced a set of scheduling annotations that describe how the module can be used. This section shows how to actually schedule rules given a module whose annotations are known. We also show how to generate the circuits that connect rules to the module interfaces. The following section then shows how to derive the module annotations and how to generate circuits for interface methods.

4.4.1 Rule validity

Before generating circuits and schedulers for a set of rules, we need to verify that each rule is valid, that is, it does not attempt to modify the same state more than once. Hence, as long as all pairs of methods in the rule have valid parallel (single rule) execution behavior, the rule is valid. More formally, if a pair of method calls $m.g_1$ and $m.g_2$ that are made inside a rule have the property that $CM_m[g_1][g_2] \in \{C, <_R, >_R, ME\}$, then the rule is not valid since the single rule execution for that pair of methods would not be defined. Technically ME is not invalid, but since it implies that the rule will never execute, we flag it as an error. Clearly, we also only need to consider method calls to the same module since we know that due to the tree structure, method calls to different modules will never interact. A final restriction on the validity check is that we only need to consider method call pairs that are not in mutually-exclusive blocks within a rule. For example, in the code below, if the compiler can derive that the two conditionals are mutually-exclusive, then it should not flag the rule as invalid, even if $m.g_1$ and $m.g_2$ are conflicting (C). Although the two methods do not have well defined single-rule behaviors, they will not be enabled simultaneously within this rule and hence it must be a valid rule.

```
rule R: when (true) =>
   if (x < 7) then
      m.g1;
   if (x > 7) then
      m.g2;
```

As described above, we apply the following *VALIDRULE?* procedure to each rule to determine its validity. *MethodCalls*(R) is assumed to return the set of method calls made by rule R.

```
VALIDRULE?(R) =
    foreach  mi.ga ∈ MethodCalls(R) do
       foreach  mj.gb ∈ (MethodCalls(R) - mi.ga) do
          if ((mi == mj) & (CMmi[ga][gb] ∈ {C, <R, >R, ME} &
               (the calls to mi.g1 and mi.g2 in R are not
                in mutally exclusive code segments))) then
             return FALSE;
    return TRUE;
```

Figure 4-11: VALIDRULE procedure

4.4.2 Rule scheduling

Next, we need to determine if each pair of rules R_1 and R_2 can be scheduled simultaneously, and if so whether there is an implied ordering constraint. Suppose we want to know if it will appear as though R_1 executes before R_2 if both rules are enabled. For this to hold, it must be true that it will appear as though every method that is called in R_1 will appear to execute before every method in R_2 executes. Thus, we start with the assumption that such scheduling is possible, and constrain the result as we examine each pair of method calls. If we encounter a method pair that does not satisfy a given ordering, then the rules will not satisfy that ordering either. We show this procedure in Figure 4-12. Again, as in the VALIDRULE procedure, we only need to consider method call pairs to the same module since we assume a tree module call structure, and only method call pairs in non-mutually-exclusive code blocks need to be considered. We use a least-upper-bound (LUB) operator as the constraining function. It is defined over the lattice of annotations in Figure 4-13. The smallest value in this lattice is CF, the largest is C.

```
DERIVEREL(R1, R2) =
   result = CF;
   foreach  mi.ga ∈ MethodCalls(R1) do
      foreach  mj.gb ∈ MethodCalls(R2) do
         if (mi == mj) then
            if (the calls mi.g1 and mi.g2 in R are not
                in mutually-exclusive code segments) then
               if (CMmi[ga][gb] == ME) then
                  return ME;
               else
                  result = LUB(result, CMmi[ga][gb]);
   return result;
```

Figure 4-12: DeriveRel procedure

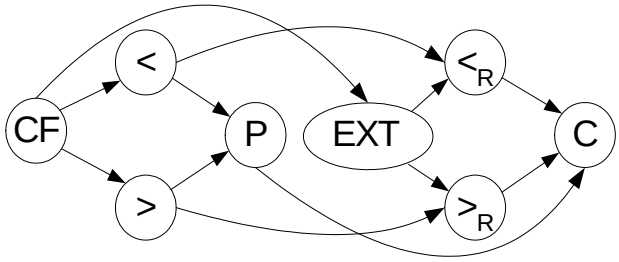


Figure 4-13: Annotation lattice

If the result of DERIVEREL(R_1, R_2) is an element of the set {CF, <, EXT, $<_R$}, then enabling R_1 and R_2 simultaneously will appear as though R_1 executes before R_2 (provided of course, the circuits to call the methods that R_1 and R_2 call are generated correctly). Similarly, if the result is an element of the set {CF, >, EXT, $>_R$}, then enabling R_1 and R_2 simultaneously will appear as though R_1 executes after R_2. As might be expected, there is some overlap in these cases—if the result is CF or EXT, then either order is possible. Thus, for each pair of rules, we can determine their sequential scheduling relationship. This is precisely the information that Hoe and Arvind's[27, 29] synthesis algorithm requires to generate a scheduler. Thus, we can derive the pair-wise rule information using the procedure above and then feed it directly into Hoe's unmodified scheduler. (Note: This works when we are just compiling rules. We will see that some modifications are required when scheduling a module's rules together with the module's methods.)

4.4.3 Rule circuit generation

In the previous section we saw how to schedule rules given only the scheduling annotations for the methods they interact with. We now show how to generate the circuits that connect the rules to the methods they call. As mentioned in Section 4.1.1, each method has a ready output signal (*_rdy*) to indicate whether its implicit condition is true, an enable input signal (*_en*) that indicates whether the method should execute, and a data input bus that is used to pass parameters. In addition, read methods have a data output bus to return their value. Circuit generation requires us to incorporate the ready signals into rule predicates, assert the enable signals, and supply input parameters for all the called methods. The outputs (results) of method calls can be fed directly into combinational logic. The circuits are generated as described in Figure 4-14. We explain each section in this circuit generation procedure in the following paragraphs.

```
Rule predicate generation:

  π_i_new        =  new rule predicate
  π_i_old        =  old rule predicate
  m_x.g_a.rdy    =  rdy signal for method call m_x.g_a in rule R_i
  p_i_mxga       =  conditional predicate of m_x.g_a call in R_i

  π_i_new = π_i_old
  foreach method call m_x.g_a in R_i do
     π_i_new = π_i_new & (m_x.g_a.rdy | ~p_i_mxga)

Method enable generation:

  m_x.g_a.en = false;
  foreach rule R_i that makes a call to m_x.g_a do
     m_x.g_a.en = m_x.g_a.en | (φ_i & p_i_mxga)

Method parameter (data) generation:

  if (DERIVEREL(m_x.g_a, m_x.g_a) == EXT) then
     m_x.g_a.data  =  parameter value that the last rule that is
                      scheduled and that calls m_x.g_a contains.
  else
     m_x.g_a.data  =  parameter value that the rule that is
                      scheduled and that calls m_x.g_a contains.
```

Figure 4-14: Modular circuit generation

Rule predicates (π_i's) are generated as follows. If a rule contains a method call whose implicit condition (ready signal) is false, then the rule must not execute, provided that the method is not located in a conditional block whose predicate is false. We prevent such rules from being scheduled by conjugating the implicit condition, along with the negation of any conditional predicate (p_{i_mxga}) with the rule's guard (π). This logic structure is required to match the behavior of the flattened design in which we had special rules for lifting of *when*'s across conditionals (see Section 3.2.2). (We use the term p_{i_mxga} to represent the predicate surrounding a call to $m_x.g_a$ in rule R_i. If the method call is not located within an *if* statement, then no such predicate exists and we set $p_{i\_mx.ga}$ to true. If multiple calls to a method occur within one rule, then we clearly need one predicate term for each method invocation. Since methods can be called at most once from each rule, at most one of these terms could be true at any time.)

The reader should recall that rule predicates feed into a scheduler (see Figure 2-6). The scheduler in turn generates a φ signal for each rule that should execute in a given cycle. Hence, if a rule executes (its φ signal is true) and it makes a call to a method $m_x.g_a$, then $m_x.g_a$'s enable signal ($m_x.g_a$.en) should be set to true, provided that the method is not called in an if statement whose predicate is false (p_{i_mxga}).

Input parameter value ($m_x.g_a$.data) generation depends on the type of method being called. If the method has an EXT annotation, then the rule that appears to execute *after* all other rules in the schedule passes the value to the method. Assuming a fixed relative scheduling ordering among rules, this can be implemented as a priority encoder. A multiplexer can be used for all non-EXT methods since each non-EXT method can be called from at most one rule in each cycle.

It should be noted that method interfaces (ports) can be viewed as resources in this scheduling / circuit generation approach. The same method cannot be called twice in the same cycle except for the EXT case. Another exception occurs when a purely combinational method is called with the same arguments in two rules. In this case, both rules can share the result of the return value.

4.5 Deriving module interface annotations

The same procedure that was used to derive the scheduling relationship among rules (DERIVEREL) can be used to determine the scheduling relationship (interface annotation) of interface methods: DERIVEREL(g_1, g_2) returns the interface annotation for methods g_1 and g_2. If we determine the relationship of all of a module's method pairs we obtain the module's CM. This CM can in turn be used to schedule modules higher in the module hierarchy. As with rules, we also need to perform a validity check on every method to ensure that it does not invoke a pair of methods that update the same state. Since we do not permit the caller to dynamically choose the order of method execution, we need to select between $<_R$ or $>_R$ in case a pair of methods can be scheduled in either order.

Using the primitive register CM in Figure 4-7, we can derive the FIFO CM by applying the DERIVEREL procedure to all FIFO method pairs. This results in the CM shown in Figure 4-15. This FIFO could then be used as a precompiled module in another design, such as the processor that we described in Figure 2-5. However, we would soon find that the annotations are more restrictive than the designer most likely intended. Since the *enq* and *deq* FIFO methods conflict, they cannot be called by two rules within the same cycle. This leads to the two processor stages not executing concurrently, that is, one stage will execute in one cycle and another stage will execute in the next. Clearly this is not satisfactory throughput. We should note that this performance problem arises regardless of whether the modular flow is used or the design had been flattened first.

$h_1 \setminus h_2$	enq	deq	clear	first
enq	C	C	$<_R$	>
deq	C	C	$<_R$	>
clear	$>_R$	$>_R$	EXT	>
first	<	<	<	CF

Figure 4-15: Derived FIFO annotations

The next chapter improves on this flow by allowing the designer to add a performance specification to the design. A design transformation algorithm will then ensure that those specifications are satisfied without altering the design's functionality. However, at this point

we can introduce a solution which is dangerous from a correctness perspective but allows the designer to utilize high-level knowledge that the compiler cannot derive. For example, the code below shows two methods which conflict since they both read state that the other rule writes. However, using high-level knowledge we know that the two methods will execute correctly if we enable them simultaneously and that their execution can be explained as execution in either order. The reason we can argue this is that we know that if a pointer is incremented in either method, it will remain positive ($x > 0$ implies $(x + 1) > 0$). (A very similar problem arises in the case of circular buffer pointers.) Generally, the compiler cannot derive such information because it relies on domain and range analysis, not a proof system that includes numerical analysis. Hence, it must assume that the two methods conflict.

```
method g1:
   cptr := cptr + 1;
when ((pptr > 0) & (cptr < 8));

method g2:
   pptr := pptr + 1;
when ((cptr > 0) & (pptr < 8));
```

A tougher case was encountered while designing the reorder buffer (ROB) of a microprocessor. Higher level logic ensured that the two simultaneous writes into the ROB could never be to the same slot but this fact is not deducible from the rule analysis without a theorem prover.

Thus, we allow the designer to override compiler-derived scheduling annotations. This is a dangerous operation since a mistake will lead to incorrect and hard-to-debug functionality, precisely what we aimed to avoid through this new synthesis process. However, for carefully crafted designs that are incorporated into pre-compiled libraries, this can be a useful feature. Because we attach annotations to modules, such assertions only have to be made at the module where the high-level knowledge is known, not at the top level as would be required in a flat synthesis flow. This limits the scope of the design that needs to be verified manually.

4.6 Module compilation

An important observation when compiling rules together with a module's interface methods is that interface methods are nearly identical to rules. The only difference is in the way they are scheduled. Rule scheduling is a local operation within a module. In contrast, methods are

scheduled external to the module. Whether or not the method executes is indicated through the method's enable signal. Thus, the enable signal can be thought of as the method's equivalent to rules' φ signals.

Surprisingly, the module interface annotations have implications for rule scheduling inside the module as well. In general a module does not know if two methods are being invoked from one rule or from two rules. The semantics must be such that the module behaves correctly in either case, provided the external scheduler is following the constraints imposed by the interface. When two methods are called from one rule then it must appear as if the external rule (together with the methods it calls) executes atomically with respect to the rules inside the module. Thus, if we do not know if the enabled methods are being called from a single or from multiple rules (because both would be valid executions), then the scheduler must assume that they are being called from a single rule and schedule all internal rules to either occur before or after the methods. Alternatively, the annotations must be restricted to not allow single-rule execution.

The above problem is best illustrated via an example. Consider the following rules and methods, where initially all registers contain 0. If R_{ext} executes followed by R_{int} we expect the result r1 = 10 and r2 = 1. If R_{int} executes followed by R_{ext} we expect the result r1 = 110 and r2 = 0. These are the only permissible outcomes for sequential execution of these two rules. However, a naïve scheduler may decide that it is alright to schedule g_1, g_2 and R_{int} with the implied ordering: $g_1 < R < g_2$. This would result in r1 = 10, r2 = 0, which violates the atomic rule execution requirements.

```
module m
   method g1:
      r1 := r1 + 100
   when (true);

   method g2:
      r2 := 0
   when (true);

   rule Rint: when (true) =>
      r1 := 10;
      r2 := r2 + 1;
endmodule
```

```
module top
   rule Rext: when (true) =>
      m.g1();
      m.g2();
endmodule
```

This problem can be avoided by either scheduling the module such that both methods appear to execute either before or after the internal rule. Alternatively, we can restrict the methods so that they cannot be invoked from a single rule, for example by changing the annotation from "<" to "$<_R$"—such restricting is always safe since we are limiting allowable behaviors, and not introducing new behaviors. This would ensure the module is used correctly, but would also invalidate the R_{ext} rule.

To complete the modular compilation flow we present the COMPILE procedure. This procedure performs a bottom-up compile. After compiling all child modules, it uses the child module annotations to generate the scheduler and circuits for the parent module's rules and methods. The GENERATESCHEDULER procedure generates a scheduler using Hoe and Arvind's scheduler generation algorithms with the additional restrictions presented in this section. The GENERATECIRCUIT procedure is equivalent to the circuit generation described in 4.4.3 (this procedure applies to both rule and method circuit generation since the process is equivalent).

```
COMPILE(m) =
   1. // Compile each module invoked by m (bottom-up)
      foreach module m_i invoked by m do
         COMPILE(m_i);
   2. // Compile the  module m
      foreach RorM_a ∈ rules and interface methods of m do
         if (!VALIDRULE?(RorM_a)) then
            return ERROR;
         foreach RorM_b ∈ rules and interfaces methods of m do
            CM_m[RorM_a][RorM_b] = DERIVEREL(RorM_a, RorM_b)
   3. GENERATESCHEDULER
   4. GENERATECIRCUIT
```

Figure 4-16: Modular COMPILE procedure

(Note: after modular rule-based compilation has been performed the design is transformed into gates using a tool such as Synopsys Design Compiler. During gate-level synthesis at least part of the design may be flattened to allow combinational optimizations across module boundaries.)

4.7 Results

Part of the impact of this modular flow is hard to quantify. For example, we do not know how many errors are avoided by using such a flow and we do not know how much time is saved by easily swapping in and out modules with different performance characteristics. Empirically, we believe it helps the design process and a flow at least partially based on these ideas is being incorporated into the Bluespec product. Others have also successfully utilized this flow to implement complex processors[11, 14].

We can quantify the advantage this flow has in compile times compared to a flow in which the design is first completely flattened. Figure 4-17 summarizes scheduling and compile time results from experimentation on several processor models. These examples illustrate the dramatic improvement in compile times that we see when using the modular flow. They also show that scheduling improves over the flat approach if we allow the designer to alter scheduling at some of the interfaces.

We worked with two ISA's—one very simple design that contains 5 instructions (5I) and one that implements a MIPS-II core[31]. The MIPS core is implemented as a fully bypassed 5-stage pipeline. In order to stress the synthesis, all designs used a complex, recursive definition of a highly-parameterized FIFO as pipeline / bypass registers. The only primitive module that was used in all designs was the primitive register. Simulations of binaries running on each processor were used to verify their functionality.

Each processor was synthesized using both the flat Bluespec flow and a modular flow. We performed modular compilation by synthesizing individual blocks and then incorporating the resulting Verilog as primitive blocks in the higher level compilation. The modular flow compiled the FIFO, and in one case also the register file (RF), as a separate module. Because of the complexity of the FIFO description, the compiler could not derive optimal annotations in the modular flow, or rule schedules in the flat flow. However, by allowing the designer to alter the annotations of the FIFO module (something that has to only be done once and can then be reused in all processor designs), we were able to achieve optimal schedules in all modular compilations.

Processor	Optimal Schedule	Partial Eval.	Scheduler	Total
5I 2-Stage Flat	No	0.7s	1.0s	3.2s
5I 2-Stage Modular	Yes	0.1s	0.1s	2.0s
5I 5-Stage Bypass Flat	No	26.8s	Opt. OFF	29.4s
5I 5-Stage Bypass Modular	Yes	0.9s	0.2s	3.6s
MIPS Flat	No	1036.1s	Opt. OFF	1052.0s
MIPS Modular FIFO	Yes	46.0s	218.1s	275.8s
MIPS Modular FIFO + RF	Yes	21.9s	1.8s	35.7s

Figure 4-17: Flat vs. modular compilation

The two largest compilation phases are partial evaluation and scheduling. The partial evaluation phase expands the code by inlining functions and modules, performs partial evaluation wherever possible, unrolls recursive calls, etc. The scheduling phase of the compiler generates the scheduler—decides which rules are mutually-exclusive, conflicting, etc. In both of these phases the modular flow is significantly faster than the flat flow. This is largely due to fewer rules needing to be compiled when using the modular flow and due to the reduction in the size of expressions. In the scheduling phase, not all optimizations could be turned on in the flat flow because expression sizes got too large for analysis, which is exponential in its runtime. As expected, the total compile time is dramatically less in the modular flow. We should note that area and timing were nearly identical in the two compilation approaches and closely matched results from a hand-coded implementation.

4.8 Possible improvements to the modular flow

There are several areas in which this modular flow can be improved. We touch on two of them in this section. The first concerns the restriction that the modular flow only applies to designs that form a tree-like module hierarchy. For many designs this is not a severe restriction. However, there are designs, such as a processor design with reorder buffer (ROB), in which it is more natural to have many modules (for example functional units) interacting with a common module (the ROB). In such cases it is possible to restructure interfaces to satisfy the tree requirement but it does not come natural to the design process. Allowing a modular compilation flow for such designs would be attractive.

In Figure 4-18 we show several non-tree structures that make modular compilation difficult. In general, the problem with such structures is that a module can no longer be compiled in isolation because its interactions with other modules depend on the other modules' behavior. For example, in the first picture of Figure 4-18, the top module does not know whether it can call the right module's method unless it knows if the bottom module is also making a call to that method. In general, these problems can only be avoided if additional glue logic is added external to the modules, something we have attempted to avoid in the compilation flow since it breaks the level of abstraction that modules are self contained.

A special case in which the modular flow can be applied to non-tree module call structures is the case in which two modules call methods of a shared module and the two methods are mutually CF. This corresponds to the second graph in Figure 4-18, with the assumption that the two methods of the module on the right are CF. Since by definition, CF methods do not interact, such a graph could be synthesized in the modular flow.

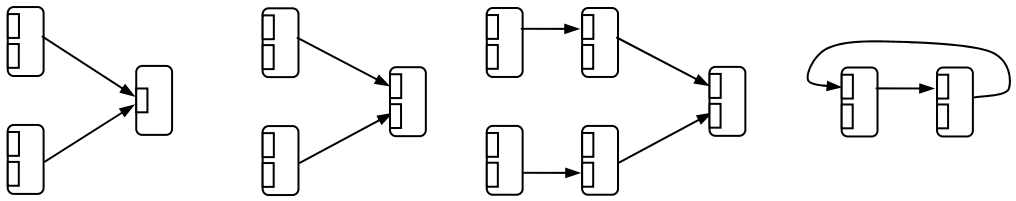


Figure 4-18: Non-tree module structure

A second problem with modular compilation is that it does not always achieve as good performance as flat compilation. For example, consider the code below. In a modular flow the compiler determines that the two methods g_1 and g_2 conflict since they could both write to the same state. Thus, it must conclude that rules R_1 and R_2 also conflict and hence cannot execute concurrently in one cycle. However, if we flatten the design and propagate constants (as shown in module top-Flattened), then the compiler would obviously conclude that the two rules do not conflict. Thus, although functionally not incorrect, the modular compilation flow does not perform as well as a flat flow in this case.

```
module m
   method g1(x):
      if (x == 0) then
         r1 := 1;
      else
         r2 := 2;
   when (true);

   method g2(x):
      if (x == 0) then
         r1 := 3;
      else
         r2 := 4;
   when (true);
endmodule
```

```
module top
   rule R1: when (true) =>
      m.g1(0);

   rule R2: when (true) =>
      m.g2(1);
endmodule
```

```
module top-Flattened
   rule R1: when (true) =>
      r1 := 1;

   rule R2: when (true) =>
      r2 := 4;
endmodule
```

In general, this type of performance penalty is rare (we have not observed it in any design yet), but is not an unusual problem to have in a modular compilation flow. Modular procedure compilation in software faces similar issues.

4.9 Chapter summary

This Chapter presented an algorithm for modular compilation of atomic actions. This compilation strategy greatly improves compile times, which in turn makes experimentation with larger designs more practical. More importantly, we believe we have introduced a flow that encourages correct-by-construction design and facilitates the exchange of reusable blocks. By incorporating semantics via the scheduling annotations into interface methods we can ensure that modules are used correctly. Furthermore, the scheduling of rule interactions with modules is handled by a compiler, which allows blocks with different schedules to easily be swapped in and out of a design. Together, we believe these contributions enhance the design flow and should make it easier to experiment with and design larger systems.

Chapter 5

Performance Specification and the EHR

Some performance guarantees in digital design are as important as correctness in the sense if they are not met we do not have an acceptable design. For example, suppose we have a pipelined processor which executes programs correctly but its various pipeline stages cannot fire concurrently because of some ultraconservative interlocking scheme. We are unlikely to accept such a design. Similarly, in the reorder buffer (ROB) of a modern 2-way superscalar processor, the designer may not feel that the design task is over until the ROB has the capability of inserting two instructions, dispatching two instructions and writing-back the results from two functional units every cycle[12]. Even simple micro-architectures (and not just related to processors) can present designers with such performance-related challenges[2]. It is important to understand that such requirements emanate from the designer of the micro-architecture as opposed to some high-level specification of the design. To that extent, only the designer can provide such specifications and they should be a core component of any high-level synthesis flow.

Bluespec relies on sophisticated scheduling of rules to achieve these goals. However, when the high-level performance goals of a designer are not met then an understanding of the schedule generated by the Bluespec compiler becomes imperative on the part of the designer before improvements can be made. This can be a challenging process. Furthermore, due to the limitations of the scheduler we have thus far described, the designer cannot always resolve

these issues without reverting to unsafe solutions, such as Bluespec Inc.'s RWire, the scheduling overrides that we introduced in the previous chapter, or other constructs that can easily introduce functional errors and make the design process substantially more difficult than we would like.

This chapter presents a new scheduling algorithm that makes user-defined scheduling constraints a core part of a design description[45, 46]. The designer can specify which rules (or methods) should execute within a cycle if their guards are satisfied and what order they should appear to execute in. These scheduling constraints are used by our new compilation flow to transform the design into a derived design which is guaranteed to be functionally equivalent to the original design and is also guaranteed to satisfy the designer's performance goals. This is a powerful and useful addition because important scheduling decisions can now be enforced by the designer rather than leaving them up to the vagaries of the compiler-generated scheduler.

We explain the new compilation flow by first introducing a well understood TRS transformation: rule composition. As we will show, rule composition is a tool that allows designs to be transformed so that they satisfy user-prescribed schedules. Finally, we address the major problem with rule composition in the context of hardware synthesis, which is that it creates an explosion of the number of rules and methods within the design. We avoid this problem by introducing a new hardware element, the *Ephemeral History Register* (EHR), along with new scheduling algorithms. These algorithms allow us to schedule rules so that it appears as though they are part of many composite rules without actually creating the compositions.

We demonstrate the power of the scheduling constraints via a simple greatest common divisor (GCD) circuit and a pipelined processor. In this chapter we show that by simply changing the performance constraints we can transform the pipelined processor into derivative designs such as an unpipelined processor, a superscalar processor, or a design with rescheduled branch resolution. These examples demonstrate that micro-architecture exploration and design specification are made much easier with the new synthesis algorithms. In the next chapter we examine the resulting circuits in more detail and show the resulting tradeoffs between scheduling (throughput) and the circuit's critical path.

We should note that the scheduling algorithms that we introduce in this chapter supplement the work described in the previous chapter. We continue to fully utilize the modular and rule-based abstractions, and rely on a modular synthesis flow to synthesize the designs after we have transformed them using this chapter's synthesis algorithms.

5.1 Understanding scheduling as rule composition

This section explains rule composition and shows how user defined schedules can be explained via rule composition. We first define rule composition and then show how it can be implemented as rules with conditional actions. We then show how rule composition can be used to satisfy scheduling constraints.

5.1.1 Rule composition

A fundamental property of rule-based descriptions is that if we add a new rule to a set of rules it can only enable new behaviors; it can never disallow any of the old behaviors. Furthermore, if the new rule being added is a so called *derived rule* then it does not add any new behaviors[5, 54]. Given two rules R_a and R_b we can generate a *composite rule* that executes R_b after R_a as follows:

```
rule Ra,b:  when (πa(s) & πb(δa(s))) =>
   s := δb(δa(s))
```

It is straightforward to construct the composed terms $\pi_b(\delta_a(s))$ and $\delta_b(\delta_a(s))$ when registers are the only state-elements and there are no modules. We illustrate this by the following two rules that describe Euclid's greatest common divisor (GCD) algorithm. They compute the GCD of two numbers by repeated subtraction and swapping of values. Note, these are the same rules we explored in Section 1.4:

```
rule Rsub: when ((x >= y) & (y != 0)) =>
   x := x - y;

rule Rswap: when ((x < y) & (y != 0)) =>
   x := y;
   y := x;
```

Using either Hoe and Arvind's compilation algorithms, or the modular algorithm from the previous chapter, we find that these two rules conflict since they both read the state that the other rule modifies. However, the designer may not want the swap to occupy an entire cycle since it does not perform much work (swapping values takes much less logic than a subtraction). A solution to this problem is to derive a new "high performance" $R_{swap,sub}$ rule that immediately performs a subtraction after a swap. By combining the swap and subtract into

one rule we reduce the number of cycles that it takes to compute the GCD, and presumably do not significantly impact the cycle time of the circuit. The derived composed rule is shown below. We name the temporary values written by R_{swap}, as x_{swap}' and y_{swap}':

```
let x_swap' = y;  y_swap' = x; in
   rule R_swap,sub :  when ((x < y) & (y != 0) &
                           (x_swap' >= y_swap') & (y_swap' != 0)) =>
      x := x_swap' - y_swap';
      y := y_swap';
```

After substitution for x_{swap}' and y_{swap}', this rule is equivalent to the following rule:

```
rule R_swap,sub:  when ((x < y) & (y != 0) &
                        (y >= x) & (x != 0)) =>
   x :=  y - x;
   y :=  x;
```

Since the $R_{swap,sub}$ rule was formed by composition it can safely be added to the GCD rule system. We can then generate a circuit for the three rules: R_{sub}, R_{swap} and $R_{swap,sub}$ using the scheduling algorithms from the previous chapter, giving preference to the $R_{swap,sub}$ rule when it is applicable. This circuit performs better than the original rule system which only contained R_{sub} and R_{swap} since it allows both the swap and subtraction to occur within a single cycle. Without composition, the scheduling analysis would not have been able to derive this parallelism and only one of the two rules would have executed each cycle. Clearly, this type of transformation improves the number of operations performed per cycle (throughput). However, it can also increase the cycle time since we are chaining operations. We analyze this issue in the next chapter.

As the GCD example shows, rule composition is an interesting tool that allows us to achieve better schedules (performance) without altering the functionality of the design. In fact, rule composition can allow us to create a composed rule for *any* set of rules, provided the rules access only registers (or primitive elements whose composition is well understood). As an example of the use of rule composition, Mieszko Lis wrote a source-to-source TRS transformation system to compose rules and applied it to a number of designs including a pipelined processor[35]. His system produced new rules by taking a cross product of all the rules in a system and filtered out those composite rules that were "uninteresting" in the following sense: Composition of R_1 followed by R_2 was considered uninteresting if either (i) it showed that R_2 could not be enabled after R_1 executed or (ii) if R_1 and R_2 were already CF or SC. In the latter case the scheduler would have scheduled them concurrently anyway and a

new rule was not required. Lis' system was able to generate all the interesting composite rules and by applying it to a simple processor pipeline's rules was able to automatically generate all the rules for a 2-way superscalar version of the processor. He was further able to show the robustness of his transformation (and filtering) by applying the transformation again to the generated 2-way rules to produce the rules for a 4-way superscalar micro-architecture. What is fascinating about this work is that it is based purely on the semantics of rule execution and does not use any knowledge specific to processor design.

The biggest problem in exploiting Lis' transformation is that in spite of his filtering of "uninteresting" composite rules, the compiler can generate a large number of new rules. He reports that the number of rules increased from 13 for the single issue pipeline to 74 for 2-issue, 409 for 3-issue, 2,442 for 4-issue and 19,055 for 5-issue pipeline[35]! These numbers reflect filtering out 24% to 41% of the possible composite rules. Although interesting from a theoretical viewpoint, this methodology is clearly not practical to generate hardware since the number of composite rules tends to grow exponentially with number of rules in the original system and the number of compositions that are performed.

Next we show how the issue of rule explosion can be avoided by introducing composition of conditional actions.

5.1.2 Rule composition using conditional actions

We now introduce conditional actions as an alternative method for rule composition. Conditional actions in rule generation subsume many natural behaviors of subsequences of rules firing, thereby dramatically reducing the number of rules that are generated during composition. Later, we show how to generate efficient circuits from these rule compositions based on conditional actions.

An example of the problem that conditional actions address is the $R_{swap,sub}$ rule that we provided earlier. This rule only covered the case when both R_{swap} and R_{sub} rules were applicable. As an alternative, consider the following rules based on conditional actions.

```
rule R_swap&sub: when (true) =>
   R_swap;   // when lifted version of R_swap
   $
   R_sub;    // when lifted version of R_sub

or:

rule R_swap&sub: when (true) =>
   if ((x < y) & (y != 0)) then
      x := y;
      y := x;
   $
   if ((x >= y) & (y != 0)) then
      x := x - y;
```

To understand the meaning of these rules we must clarify why we replaced the rules' *when* clauses by *if* statements, and what it means for a rule body to contain a "$". Replacing the rules' *when*'s by *if*'s is performed simply via *when* lifting as described in Section 3.2.2 (after *when*'s have been lifted a *when* statement is equivalent to an *if*):

when (p_1) => a_1; $\Leftrightarrow$ **if** $(p_1$ & $a_{1\_cond})$ **then** a_{1_body};

In these rules, the meaning of "$" is the same as we introduced in 3.3.2, that is, the actions following the "$" see the effect of actions before the "$". More formally we said, given an initial state *S*, and rules R_a and R_b, we obtain the result state S_{next} when evaluating R_a \$ R_b:

$S_{next} = E_{Rule}\ [\![R_a\ \$\ R_b]\!]\ S$

Is equivalent to:

$S' = E_{Rule}\ [\![R_a]\!]\ S$
$S_{next} = E_{Rule}\ [\![R_b]\!]\ \mathbf{S'}$

This new rule has the advantage over standard composition in that it behaves as rule R_{swap} if rule R_{sub} does not get enabled; it behaves as rule R_{sub} if rule R_{swap} does not get enabled and behaves as R_{swap} followed by R_{sub} if R_{swap} is enabled and that in turn does not disable R_{sub}. Hence, based on conditional actions, we have generated a single rule that behaves as three rules: R_{sub}, R_{swap}, and the derived composed rule $R_{swap,sub}$. For *n* rules, this approach introduces at worst one rule consisting of *n* conditional actions, whereas traditional composition introduces an exponential number of new rules during composition.

Previous synthesis and scheduling algorithms cannot compile rules and methods that contain a "$" since there was no notion of sequencing within a rule. However, with appropriate

renaming we can derive an equivalent rule which eliminates the "$" from the rule. We illustrate this again via the GCD example. The basic idea in this renaming scheme is that we read the initial values of x and y into x^0 and y^0. We then compute the next state values for actions before the "$" ($x^1$, y^1), then compute the values for the actions following the "$" ($x^2$, y^2), using x^1 and y^1 in place of x and y. Finally, we assign the last values (x^2 and y^2) to x and y:

```
rule R_swap&sub: when (true) =>
   // initialize x^0 and y^0
   let
      x^0  = x;
      y^0  = y;
      // swap if the swap predicate is true
      x^1  = ((x^0 < y^0) & (y^0 != 0)) ? y^0 : x^0;
      y^1  = ((x^0 < y^0) & (y^0 != 0)) ? x^0 : y^0;
      // subtract if the sub predicate is true
      x^2  = ((x^1 >= y^1) & (y^1 != 0)) ? x^1 - y^1 : x^1;
      y^2  = ((x^1 >= y^1) & (y^1 != 0)) ? y^1 : y^1;
      // update the registers
   in
      x  := x^2;
      y  := x^2;
```

This rule does not contain a "$", but simply a set of combinational assignments (to x^i's and y^i's) and two actions (writes to x and y). Thus, this rule is no different from the types of rules we have discussed in previous chapters and could be compiled together with other rules using the previously described rule synthesis. It should also be clear that it behaves exactly like the composed rule that contained the "$".

Thus, if rules only interact with primitive registers, we can construct rules that implement many subsequences of rule composition by first creating composite rules that contain conditional actions and then removing the conditional actions via a renaming step. In the next subsection we show how we can use these styles of rules to satisfy scheduling constraints.

5.1.3 Performance constraints

The goal in this chapter is to allow a designer to specify a set of rules that should be scheduled together if their guards are true. Additionally, we will show that it is important for the designer to be able to specify in what sequential order the rules should appear to execute. This

subsection shows how constraints are specified and how they can be directly translated into conditional actions.

An example of a constraint for the GCD program is shown below:

$R_{swap} < R_{sub}$

This constraint specifies that R_{swap} and R_{sub} should both execute within a cycle if their guards are true and that it should appear as though R_{swap} executes first. If only one of the rules is enabled, then that rule should execute.

More generally, we can specify constraints for multiple rules. Each such guarantee (ARPG—Arvind Rosenband Performance Guarantee) takes the following form:

```
ARPGs                ::=    [ARPG]

ARPG                 ::=    <Performance Group> "<"
                            <Performance Group> "<"
                            …

Performance Group    ::=    {Ra, Rb, Rc, …}
```

Figure 5-1: ARPG syntax

A design can contain multiple performance guarantees and we refer to each set $\{R_a, R_b, \ldots\}$ within a guarantee as a performance group. Although not a strict requirement, to facilitate better understanding of what these constraints mean we will assume that all the rules and methods in a performance group are either pair-wise ME or CF.

The idea behind the guarantee is that after transforming the design, all rules (methods) in the guarantee can be scheduled together. Additionally, if performance group S_i appears before S_j in the guarantee, then any enabled rule (method) from S_i will appear to execute before any enabled rule (method) in S_j. Also, as mentioned earlier, all subsets of rules within a performance guarantee should execute, even if other rules in the guarantee are not enabled.

It turns out that we can transform ARPG's directly into conditional actions. Suppose we are given a set of rules R_1, R_2, R_3, etc:

```
rule R1: when (p1) => a1;
rule R2: when (p2) => a2;
rule R3: when (p3) => a3;
```

Then, we can directly translate any constraint:

$R_1 < R_2 < R_3 < \ldots$

into a sequential rule:

```
rule R1,2,3: when (True) =>
    R1;
    $
    R2;
    $
    R3;
    $
    …
```

These rules satisfy the "sequential" and "subset" scheduling properties—they appear to execute in the order specified by the ARPG and any subset of rules will execute if enabled. Thus, given any constraint, we can create a composed rule that satisfies the constraint while preserving the functionality of the original design. If multiple ARPG's are provided, we construct a sequential rule for each one of them. However, the question remains, how to generate circuits from arbitrary rules that contain sequential actions ("$"'s). We explore this in the next section for rules that contain method calls to registers only, and extend the ideas to methods in the following section.

5.2 Transforming composed rules

The definition of "<" in Section 4.2 states that there is no difference between "$" composition (sequential) and ";" composition (parallel) if the rules satisfy the "<" property:

$$\text{if } P_0 < P_1 \text{ then } P_0 \$ P_1 \equiv P_0 \; ; \; P_1$$

Now suppose P_0 and P_1 do not satisfy the "<" property. Can we derive P_0' and P_1' such that (i) $P_0 \$ P_1 \equiv P_0' \$ P_1'$ and (ii) $P_0' < P_1'$, and hence $P_0' \$ P_1' \equiv P_0' \; ; \; P_1'$? If both these conditions are satisfied, then $P_0 \$ P_1 \equiv P_0' \; ; \; P_1'$ must be true and we have shown how to eliminate the "$" from a rule. We show this in two steps:

(1) We show how to generate P_0' and P_1' if the only method calls in P_0 and P_1 are to registers.

(2) We show how to generate P_0' and P_1' if P_0 and P_1 make calls to arbitrary interface methods.

5.2.1 Composition of rules with only register method calls

In Section 5.1.2 the GCD example demonstrated that via renaming it is possible to transform two sequential rules into a single conventional rule. Here we show how such a transformation can be systematically accomplished. The goal is to take a composite rule that contains "$'s", and replace it with a functionally equivalent rule where all "$'s" have been removed. This new rule can then be synthesized using the standard rule-based synthesis algorithms.

The basic idea in this transformation is that we rename state accesses. However, unlike the renaming process in the previously described GCD example we do not introduce new rename variables inside each rule. Instead, we rename the method calls that interact with the state elements (in this section registers only) and rely on the lower-level module to implement the variable renaming step. As we will see, renaming interface methods instead of the actual state variables has the advantage that in a module hierarchy a rule does not need to have direct access to the modules internals to perform renaming.

Below we repeat the composed GCD rule with explicit read and write method calls:

```
rule R_swap&sub: when (true) =>
   if ((x.read() < y.read()) & (y.read() != 0)) then
      x.write(y.read());
      y.write(x.read());
   $
   if ((x.read() >= y.read()) & (y.read() != 0)) then
      x.write(x.read() - y.read());
```

Now, suppose we numbered the method calls before the "$" to have index 0, and those after the "$" to have index 1. (We indicate a method's index via a superscript.). The resulting rule is shown below:

```
rule R_swap&sub: when (true) =>
   if ((x.read⁰() < y.read⁰()) & (y.read⁰() != 0)) then
      x.write⁰(y.read⁰());
      y.write⁰(x.read⁰());
   $
   if ((x.read¹() >= y.read¹()) & (y.read¹() != 0)) then
      x.write¹(x.read¹() - y.read¹());
```

This transformation is correct, provided the following conditions are satisfied. These conditions state that $read^i$ and $write^i$ behave precisely like the standard register *read* and *write* method calls if no *read* or *write* with index other than *i* is called. They do not say anything about the relationship of method calls with different indices.

```
1a)   r.read^i returns the current state of r
1b)   r.write^i(v) changes r to have the value v
2)    r.read^i < r.write^i
```

One possible choice of method implementation is to have *read*^{i} simply call the register *read* method, and *write*^{i} the register *write* method. Although correct, such a choice of methods would not accomplish anything since they would not help us eliminate the "$". Instead, to eliminate the "$" we need to satisfy the additional conditions for i < j:

```
3a)   r.read^i < r.read^j
3b)   r.read^i < r.write^j
3c)   r.write^i < r.read^j
3d)   r.write^i < r.write^j
```

If these conditions are satisfied, then we can eliminate the "$". The reason this is possible is that by construction all statements before a "$" have a lower index than the statements after the "$". By the above restrictions, this implies that all method calls before the "$" are "<" the method calls that occur after the "$". By the theorem in the previous subsection, we can then eliminate the "$". To complete the example, we show the resulting GCD code below:

```
rule R_swap&sub: when (true) =>
   if ((x.read^0() < y.read^0()) & (y.read^0() != 0)) then
        x.write^0(y.read^0());
        y.write^0(x.read^0());
   if ((x.read^1() >= y.read^1()) & (y.read^1() != 0)) then
        x.write^1(x.read^1() - y.read^1());
```

We can summarize the above transformation algorithm. This procedure eliminates the "$" from a composite rule that only references registers without altering its behavior:

```
Given a rule R:

Let $loc be a function that returns the location in the "$"
sequence for each method call.  That is, given a $ b $ c $ …,
$loc is defined such that $loc(a) = 0, $loc(b) = 1, etc.

1)  foreach method call m.h in R do
       set the index of m.h to $loc(m.h)

2)  Remove all $'s from R
```

Figure 5-2: Method indexing procedure

Thus far we have shown how to transform a composite rule into a "normal" rule, provided that the rule only makes calls to the primitive register element. However, this transformation relies on a new register state element that satisfies the 7 conditions listed earlier in this subsection. The next subsection introduces a new state element, the Ephemeral History Register (EHR), which satisfies these conditions. Once this register has been introduced we have a complete flow to generate composite rules that interact with registers only. We will then extend the algorithms to apply to arbitrary method calls.

5.2.2 The Ephemeral History Register (EHR)

The Ephemeral History Register (EHR)[45, 46] is a new primitive state element that supports the forwarding of values from one rule to another. It is called Ephemeral History Register because it maintains a history of all writes that occur to the register within a clock cycle. Each of the values that were written (the history) can be read through one of the read interfaces. However, the history is lost at the beginning of the next cycle.

The circuit for this new primitive state element is shown in Figure 5-3. As in a conventional register, each *read* method returns a value, and each *write* method has an enable input signal (en) and a data input value (x).

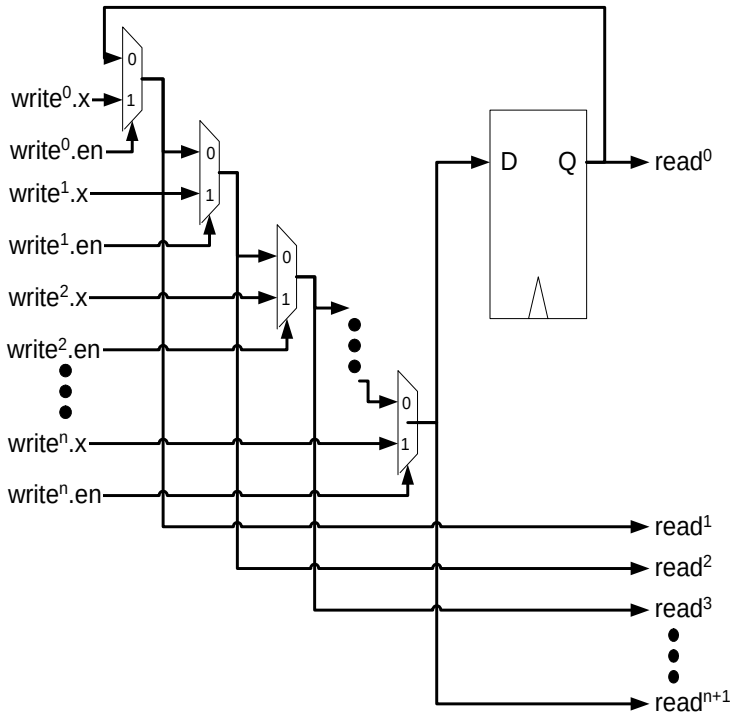


Figure 5-3: The Ephemeral History Register

It is clear that we can use the EHR in place of a standard primitive register element by replacing calls to the register *read* and *write* methods with calls to the EHR $read^0$ and $write^0$ methods. These interfaces behave exactly as those of a normal register if none of the other interfaces are being used. Similarly, any pair of methods $read^i$ and $write^j$ behaves like the conventional register, provided no other methods are called. In addition, $read^i$ does not observe the value written by $write^i$, and hence $read^i < write^i$ must hold. Thus, the EHR satisfies the first three (1a, 1b, and 2) of the conditions in Section 5.2.1 that our new register must satisfy.

The more interesting cases arise when EHR methods with different indices are enabled. Each $read^i$ returns the value written by the $write^j$ method, where j satisfies the properties that: $j < i$, and no $write^k$ for $j < k < i$ is enabled. (As a reminder, if a method is not called, then its enable signal is always false.) If no such write takes place, then the $read^i$ method returns the current state of the register. With regards to the next state of the register, the *write* method that is enabled with the largest index takes precedence over all other writes, that is, it determines the value of the state element will contain in the next cycle. If no *write* method is enabled, then the state does not change. From these observations we can conclude that the EHR satisfies the remaining four constraints of Section 5.2.1.

For completeness we show the conflict matrix for the EHR in Figure 5-4. This conflict matrix is not derived by a compiler but provided as a new primitive conflict matrix. However, rules or methods that interact with the EHR can be synthesized using this conflict matrix in the modular compilation flow from the previous chapter. (Note: we only provide the conflict matrix for *read* and *write* methods with index 0 and 1. However, it should be clear how this extends to higher indices. It should also be apparent that the EHR circuit structure can be extended to include methods with arbitrarily large indices.)

	$read^0$	$write^0$	$read^1$	$write^1$	…
$read^0$	CF	<	<	<	…
$write^0$	>	EXT	<	<	…
$read^1$	>	>	CF	<	…
$write^1$	>	>	>	EXT	…
…	…	…	…	…	…

Figure 5-4: EHR conflict matrix

It is also worth noting that the EHR effectively implements renaming via its interface methods. Given a variable *x*, we can think of $x.read^0$ as reading x^0, $x.write^0$ writing x^1, $x.read^1$ reading x^1, etc. However, rather than exposing these variables and hence the module's internals directly, we accomplish the same effect via renaming of the interface methods.

In summary, using the rule transformation algorithm in Figure 5-2 and the EHR as the new state element we can generate arbitrary composite rules for rules that interact with registers only. Next we show how these algorithms can be extended to arbitrary modules.

5.3 Modular composition

The previous sections showed how to compose arbitrary rules that interact with registers only. This section extends the algorithms to rules and methods that interact with arbitrary interface methods. The goal is the same as it was with registers: to transform P_0 \$ P_1 into P_0'; P_1', by creating P_0' and P_1' such that they behave individually as P_0 and P_1, and such that their relationship is P_0' < P_1'. Our approach is to rename method calls (give them an index) and rewrite the interface methods so that the renamed (indexed) methods satisfy certain properties—similar to the properties that the EHR *read* and *write* methods satisfy. We first provide properties that the indexed methods must satisfy. We then show that assuming that these properties hold, that it is straightforward to create the transformed programs P_0' and P_1'. Finally, we present an algorithm for creating the indexed methods and prove that the resulting methods satisfy the desired properties.

Let us assume that for each module *m* and all interface methods *m.g* and *m.h*, we can create new interfaces with the following properties:

```
MP1)  m.g^i behaves the same as m.g
MP2)  m.g * m.h => m.g^i * m.h^i   (where * ∈ {<, >, C, ...})
MP3)  m.g^i < m.h^(i+1)
```

The first property (MP1) says that if $m.g^i$ is the only method of module *m* that is enabled, then it must behave exactly as though the original method, *m.g*, was enabled. The second condition (MP2) states that the relationship between any two methods with the same index ($m.g^i$ and $m.h^i$) must be the same as the relationship of the original methods (*m.g* and *m.h*). This means that if any non-conflicting subset of methods with the same index is called, then the behavior must be exactly the same as the same subset of the original methods.

Property MP3 states that if two methods are called where the second methods has a higher index than the first method, then the behavior must be explainable as the sequential execution of the two methods such that the lower indexed method appears to execute first. We should note that these are generalized constraints for the constraints that we imposed on the EHR implementation in the previous section and in 5.2.1.

Assuming we can construct new interface methods that satisfy the above conditions, we can safely apply the following transformations to the programs P_0 and P_1:

```
T1) Given a program P in which all method calls have the same
    index (i):
       Replace every method call in P by a method call with
       index j

T2)  Given P0 $ P1 where the index of all method calls in P0 is
     less than the index in P:
       Replace P0 $ P1 by P0; P1
```

Let us understand why these transformations are valid. Transformation T1 is valid because of interface method properties MP1 and MP2. By uniformly changing the index of method calls we do not alter the behavior of a program because (i) the methods themselves do not change their behavior (MP1), and (ii) simultaneous execution of the newly indexed methods is explainable as simultaneous execution of the original methods (MP2).

Transformation T2 is explainable by property MP3. If the indices of all method calls in P_0 are less than the indices of the method calls in P_1 then by MP3, all method calls in P_0 must be < the method calls in P_1. Hence, $P_0 < P_1$ must hold. Furthermore, we showed earlier that if $P_0 < P_1$, then $P_0 \$ P_1 \equiv P_0; P_1$. Thus, T2 must also be a valid transformation. (Note: since we assume a tree module call hierarchy, methods of different modules are automatically <.)

Thus, if we could produce indexed methods with the above properties, then we could transform any composite rule that contains a "$" into an equivalent rule in which the "$" has been eliminated. Such a rule could then be compiled using the standard rule-based synthesis algorithms. The procedure to eliminate the "$" is precisely the same procedure we used for the register only case, see Figure 5-2: we would change all method calls in P_0 to be method calls to methods of index 0 (apply T1), and all method calls in P_1 to methods of index 1 (apply T1)—resulting in P_0' and P_1'. Since all indices in P_1' are then higher than those in P_0' we can replace P_0' \$ P_1' by P_0'; P_1'; (apply T2).

Using the above procedure we can eliminate the "*$*" from any sequential rule. The only step that remains to complete the algorithm is to show how to generate indexed methods that satisfy properties MP1, MP2, and MP3. A surprisingly simple procedure can be used to create the indexed methods:

```
Procedure to create m.g^i from m.g:

Rename all method calls inside method m.g^i to be calls to
methods with index i.
```

Figure 5-5: Method renaming procedure

We can use an inductive proof to show that such renaming satisfies properties MP1, MP2, and MP3. The proof occurs over modules in the call hierarchy. That is, we show that the indexing algorithm satisfies the desired properties, provided that the renaming/indexing scheme satisfies the properties for all methods that the module calls:

Base Case: By design the EHR satisfies the conditions MP1, MP2, and MP3. These properties are generalized properties of the conditions 1-7 that we used to create the EHR.

Inductive Hypothesis: The properties hold for all interface methods that the methods of module *m* call.

Inductive Proof:

We need to show that given the inductive hypothesis that the renaming of module *m*'s methods satisfies MP1, MP2, and MP3.

Property MP1: By the inductive hypothesis we know that all methods that *m.g* calls must satisfy property MP1 and MP2. Thus, if we call methods of index *i* rather than the original index-less methods in *m.g*, the behavior must not change. Hence, $m.g^i$ behaves the same as *m.g*.

Property MP2: Suppose *m'.a* is called in *m.g* and *m'.b* is called in *m.h*. By the inductive hypothesis we know that all methods that *m.g* and *m.h* call must satisfy property MP2. This means that if we enable *m'.a* and *m'.b* simultaneously then the behavior is the same as if we enable $m'.a^i$ and $m'.b^i$ simultaneously. Hence the behavior of enabling $m.g^i$ and $m.h^i$

simultaneously must be the same as enabling *m.g* and *m.h* together. This is what property MP2 states.

Property MP3: Suppose *m'.a* is called in method *m.g* and *m'.b* is called in method *m.h*. By the inductive hypothesis we know that all methods that *m.g* and *m.h* call must satisfy property MP3. This means that if we enable $m'.a^i$ and $m'.b^{i+1}$ simultaneously then the behavior is the same as if we executed $m.a^i$ and $m.b^i$ in sequence. Hence the behavior of enabling $m.g^i$ and $m.h^{i+1}$ simultaneously must be the same as executing *m.g* followed by *m.h*. This is what property MP3 states.

□

5.4 Performance driven composition algorithm

This section combines the ideas from the previous sections to create an algorithm which accepts as input a design and performance constraints, and produces as output a derived design which is functionally equivalent to the original, but is also guaranteed to satisfy the performance guarantees. This algorithm can be performed by hand or implemented as an intermediate pass in the Bluespec compilation flow. As defined by the ARPG syntax in Figure 5-1, each scheduling constraint *C* that is provided as input to this algorithm takes the form $S_0 < S_1 < S_2 < \ldots$, where each S_i is a set of rules or methods.

As shown in Figure 5-6, the algorithm can be divided into three steps. First we assign a set of indices to each rule and method that appears in the scheduling constraints—if a rule or method appears in S_i, then we add index *i* to that rule or method. Unconstrained rules are assigned index 0. Next we propagate the indices through the module hierarchy. The idea in this step is that if a rule or method has an index *i* assigned to it and makes a call to a method *m.h*, then the indexed method $m.h^i$ will need to be available in the next step—hence, we assign index *i* to the method *m.h*. We continue to propagate these indices through the hierarchy until a fixed point is reached. The final step creates the indexed rules, methods and local bindings. This process involves replicating the program component being indexed, and applying a renaming procedure as described in Figure 5-5. Since we cannot propagate into leaf nodes, we must also replace all registers with EHR's so that indexed read and write methods become available.

```
// note:  we abbreviate "all rules, methods, or local
// bindings" as RorMorLB, we abbreviate "all rules or methods"
// as RorM, etc.

PERFORMANCESCHEDULE(C0, C1, …)
0)  initialize the set of indices assigned to each rule,
    method and local binding to be empty.
 foreach RorMorLB in the design do
    indices[RorMorLB] = ∅;

1)  Assign indices to rules, methods and local bindings based
    on the constraints.  Unconstrained rules are assigned
    index 0.
 foreach Ci do
    foreach Sj in Ci do
       foreach rule or method RorM in Sj do
          indices[RorM] = indices[RorM] ∪ j;
 foreach rule R in the design do
    if (indices[R] == ∅) then
       indices[R] = 0;

2)  Propagate indices through the module hierarchy
 while a fixed point has not been reached do
    foreach RorMorLB in the design do
       foreach MorLB that RorMorLB references do
          indices[MorLB] = indices[MorLB] ∪ indices[RorMorLB];

3)  Create indexed rules, methods, and local bindings
 foreach RorMorLB in the design do
    foreach i ∈ indices[RorMorLB] do
       Create a copy (RorMorLB') of RorMorLB;
       foreach MorLB referenced in RorMorLB' do
          replace MorLB with a reference to MorLB^i;

 Replace all reg's by EHR's;
```

Figure 5-6: Performance driven scheduling algorithm

In the next sections we present two examples of how this procedure is applied to designs. However, from the analysis in the previous sections it should be clear that the resulting designs always satisfy the desired performance constraints. In addition, none of the transformations alter the functional behavior of the design. After the transformations have been performed we can generate the design's circuit implementation using the modular synthesis flow from the previous chapter.

A final step that can be added to the performance driven scheduling algorithm is a pruning procedure. The motivation for this step is that the performance driven scheduling procedure can result in references to EHR methods that are larger than required. For example, suppose R_3, as part of a sequence $R_0 < R_1 < R_2 < R_3$, is the only rule to access a register reg_{only3}. The algorithm turns reg_{only3} into an EHR and provides R_3 access to it via interfaces $read^3$ and $write^3$. However, since none of the other rules access the ports 0, 1, or 2 of the register reg_{only3} it is wasteful to have R_3 tap the register at such a high index number. It could simply have accessed the register through the $read^0$ and $write^0$ interfaces. Thus, we should run the PRUNE procedure in Figure 5-7 after running the PERFORMANCESCHEDULE algorithm. (Note: we do not perform PRUNE's in the examples in the next sections because it convolutes the numbering that occurs in the PERFORMANCESCHEDULE algorithm.)

```
PRUNE() =
 foreach EHR (r) in the design do
   while (ports in r can be pruned) do
     if r.read^i is used and r.write^(i-1) is unused then
       change the use of r.read^i to r.read^(i-1)
     if r.write^i is used and r.read^i is unused
                        and r.write^(i-1) is unused then
       change the use of r.write^i to r.write^(i-1)
```

Figure 5-7: PRUNE procedure

5.5 Specifying schedules for a pipelined processor

This section demonstrates the power of the performance scheduling algorithm via a 4-stage pipelined processor. We use a simple processor with only two instructions: Add and Jz (branch on zero). These instructions contain the interesting scheduling issues that arise in a larger processor. However, by using only two instructions we can focus on the scheduling problems rather than the details of each instruction.

We show the processor pipeline code in Figure 5-8 and a processor pipeline diagram in Figure 5-9. The processor stages are connected by FIFO buffers *bF*, *bD* and *bE*. In addition to the usual *enq, deq, clear,* and *first* methods, the *bD* and *bE* FIFO's also provide a *bypass* method to search the FIFO for a particular destination register and return the associated value. (Note: *bypass* returns a pair: *matches* is true if a match is found; *value* contains the associated value if it is found.) The processor has a total of 7 rules: *F* fetches an instruction and puts it in

bF; *D_add* and *D_jz* decode the first instruction in *bF* and fetch the operands either from the register file or the bypass path and *enqueue* the decoded instruction into *bD*; the *E* rules execute the first instruction in *bD* and either enqueue the results in *bE* or, in case of a branch taken, clear the fetched instructions from *bF* and *bD*; *WB* writes back the value in the register file.

```
function stall(src) =
 {matches, value} = bD.bypass(src);
 return matches;

function bypassv(src) =
 {matches, value} = bE.bypass(src);
 if (matches) then
    return value;
 else
    return rf.read(src);

rule F: when (true) =>
 bF.enq(imem[pc]);
 pc := pc + 4;

rule D_add: when  (bF.first() == (Add rd ra rb)) &
                  (!stall(ra)) & (!stall(rb)) =>
 bD.enq(EAdd rd bypassv(ra) bypassv(rb));
 bF.deq();

rule D_jz: when   (bF.first() == (Jz cd addr)) &
                  (!stall(cd)) & (!stall(addr)) =>
 bD.enq(EJz bypassv(cd) bypassv(addr));
 bF.deq();

rule E_add: when (bD.first() == (EAdd rd va vb)) =>
 bE.enq(WB rd (va + vb));
 bD.deq();

rule E_jz_taken: when ((bD.first() == (EJz cd av)) &
                       (cd == 0))  =>
 pc := av;
 bD.clear();
 bF.clear();

rule E_jz_nottaken: when ((bD.first() == (EJz cd av)) &
                          (cd != 0)) =>
 bD.deq();

rule WB: when (true) =>
 rf[bE.first().rd] = rf[bE.first().val]
 bE.deq();
```

Figure 5-8: 4-stage processor code

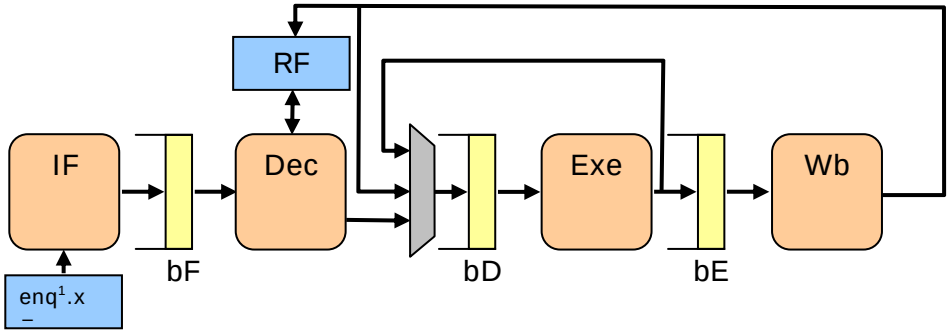


Figure 5-9: 4-stage processor pipeline

The FIFO code is shown in Figure 5-10. We use the same style FIFO as in Figure 4-2, except this FIFO contains only a single-element and now also includes the bypass logic.

```
module FIFO
  // local state definition
  mkReg data0;          // contents of FIFO element 0
  mkReg full0;          // 1 if FIFO element 0 contains valid
                        // data, 0 otherwise
  // interface specification
  method enq(x) =
     full0 := 1;
     data0 := x;
  when (full0 == 0);    // to enq, FIFO must no be full

  method deq =
     full0 := 0;
  when (full0 == 1);    // to deq, FIFO must not be empty

  method clear =
     full0 := 0;
  when (true);          // can be called anytime

  method first =
     return data0;      // return the first FIFO element
  when (full0 == 1);    // FIFO must contain valid data

  method bypass(src) =
     return {.matches = full & (data0.rd == src),
             .val     = data0.val};
  when (true);          // FIFO must contain valid data
endmodule
```

Figure 5-10: Single-element FIFO with bypass

We should recall that the modular rule-based design flow is attractive for this style of design because it allows us to focus on each pipeline stage without needing to consider what the other stages are doing at the same time. For example, both the *E_jz_taken* and *F* rules update the *pc*. However, we can consider each of these rules in isolation. If each rule behaves correctly, then the execution model ensures that correct behavior will be observed in a system that includes both rules. Similarly, when we design the FIFO, we can focus on the implementation of each method and do not need to ask such questions as what happens when *enq* and *deq* are called simultaneously.

Although attractive from a design flow perspective, we saw in the previous chapter that this pipeline has performance (throughput) problems: The modular synthesis flow determines that the FIFO *enq* and *deq* cannot execute simultaneously. Hence, consecutive pipeline stages cannot execute within the same cycle. Although still functionally correct, most designers would be dissatisfied with this result. Now, we can see how performance constraints help solve this problem.

For this processor to behave as a conventional pipeline, all rules must execute concurrently when enabled. In addition, an ordering is required such that it appears as though the *WB* rule executes followed by the *E* rules, followed by the *D* rules, followed by the *F* rule in each cycle. Additionally, if any of the stages cannot execute, for example due to a stall condition, then if possible, the remaining subset of rules should continue to execute. This can be written as an ARPG as follows:

```
{WB rule} < {E rules} < {D rules} < {F rule}
```

There are several reasons this ordering is important. Since we are using a single-element FIFO as a pipeline stage, a value needs to be dequeued before a new value can be enqueued. Hence, a rule that dequeues must appear to execute before the rule that enqueues in the previous pipeline stage. Similarly, the execute rule must appear to execute before the decode rule since our expectation is that result values can be forwarded from the execute rule to the decode rule via a bypass path.

If we apply the PERFORMANCESCHEDULE procedure to the processor design with the above ARPG as the input constraint we obtain a design that behaves with the expected performance. To better understand this, we walk through part of the procedure's execution. The first step assigns indices to all rules that appear in the ARPG. In this case, the *WB* rule is assigned index 0, the *E* rules are assigned index 1, the *D* rules are assigned index 2, and the *F*

rule is assigned index 3. Next, the indices are propagated through the module hierarchy. Examining the *bE* FIFO, we see that the following indices are assigned to its methods. In this table the first column indicates the method name, the second column contains the index assigned to it, and the third column shows which rule caused that index number to be assigned to it.

Method name	Index number	Who assigned the index?
enq	1	E rules
deq	0	WB rule
clear	/	/
first	0	WB rule
bypass	2	D rules

After the above indices are propagated into the FIFO methods, and the registers have been replaced by EHR's, we obtain the FIFO code shown in Figure 5-11.

```
module FIFO
  // local state definition
  mkEHR data0;          // contents of FIFO element 0
  mkEHR full0;          // 1 if FIFO element 0 contains valid
                        // data, 0 otherwise
  // interface specification
  method enq[1](x) =
     full0.write[1](1);
     data0.write[1](x);
  when (full0.read[1]() == 0);

  method deq[0] =
     full0.write[0](0);
  when (full0.read[0]() == 1);

  method first[0] =
     return data0.read[0]();
  when (full0.read[0]() == 1);

  method bypass[2](src) =
     return {.matches = (data0.read[2]().rd == src),
             .val     = data0.read[2]().val};
  when (true);
endmodule
```

Figure 5-11: bE FIFO

If we then apply the modular synthesis algorithm from the previous chapter to this FIFO we obtain the following conflict matrix. (Note: we use the EHR conflict matrix from Figure 5-4 in this process.)

	first[0]	deq[0]	enq[1]	bypass[2]
first[0]	CF	<	<	<
deq[0]	>	C	<	<
enq[1]	>	>	C	<
bypass[2]	>	>	>	CF

As we expect, by construction all lower indexed methods are < the higher indexed methods. Most notably, $deq^0 < enq^1$ and $enq^1 < bypass^2$. These two annotations indicate that the pipeline stages can now execute concurrently and that the bypass logic observes the latest value being enqueued into the FIFO. Similar transformations apply to the other FIFO's and the register file in the processor. The result is a processor whose pipeline executes with the expected throughput.

Other schedulers can be applied to the same processor design to obtain interesting behaviors. For example, if we replace the single-element FIFO with a two-element FIFO, and apply the following scheduling constraint, we obtain a two-way superscalar processor:

```
WB < WB < E < E < D < D < F < F
```

This schedule says write back two instructions one after another, execute two instructions one after another, decode two instructions one after another and fetch two instructions one after another—all in one clock cycle. This is precisely the way a two-way superscalar processor is supposed to function[25]. It should not come as a surprise that if the machine has to actually behave like a two-way issue machine then it would need more resources. Indeed we would see that implementing this schedule would require more interfaces on the FIFO's and register files and, and more combinational logic to implement two copies of the original rules in each pipeline stage.

A final processor schedule that could be interesting is shown below:

```
F < D < E < WB
```

This schedule transforms the processor into a single cycle (unpipelined) processor. The reason for this is that the fetch rule would first enqueue a value into the *bF* FIFO. The decode rule

would then dequeue the value (within the same cycle), process it, and enqueue it into the *bD* FIFO. Instructions would continue to "fly" through the pipeline until they are written back in the *WB* stage.

Thus far we have shown how we to transform a processor pipeline to satisfy different scheduling and throughput requirements using the PERFORMANCESCHEDULE algorithm. The next chapter examines the circuits that result from this procedure and shows that in most cases they are precisely what the designer expects.

5.6 Mixed rule and method constraints

The previous section showed how to propagate constraints on top-level rules through a module-hierarchy. This section presents a simple blocking cache design in which we mix constraints on both methods and rules. As a designer, we like to think of the cache problem as discreet events as shown in Figure 5-12: 1) we receive a cache request, 2) we check the cache to see if there is a hit or miss—if there is a hit we enqueue the result into the reply queue, otherwise send a request to the main memory, 3) accept the memory reply if required, and 4) return the result.

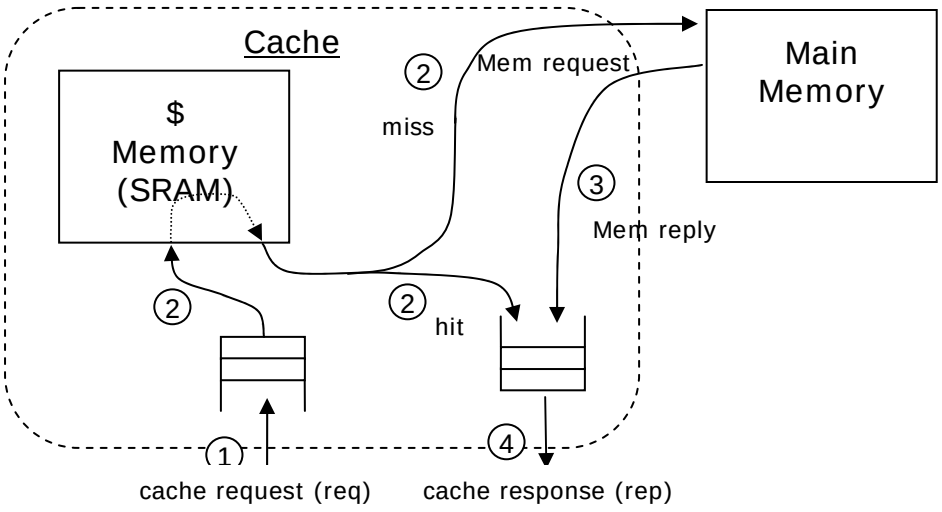


Figure 5-12: Cache block diagram

Rule-based design allows us to write each of these events as its own method or rule. This is shown in Figure 5-13. This code hides many of the logic details, for example we assume a *cachehit* function exists which returns true if a given address hits in the cache. However, all these functions are purely combinational.

If we synthesize this design without performance constraints, a cache hit requires three cycles (step 1, 2, and 4). This might be acceptable, or even the desired behavior. However, some designs might require a single cycle cache-hit performance. In a conventional flow this would require a redesign of the block. However, using performance constraints, we can transform the pipelining such that the stages are removed if a cache hit occurs. The constraint that achieves this effect is:

```
cache_req < {hit, miss} < cache_resp
```

In this case we have mixed scheduling of rules and interface methods. Furthermore, we have scheduled rules to appear to execute in between two methods calls. As pointed out in Section 4.6, this can be dangerous since we cannot always guarantee atomicity if this happens. To ensure atomicity, we must not allow the methods to be called from a single rule. This can be accomplished by restricting the relationship between methods to be $<_R$ in all cases in which performance constraints schedule rules in between the methods.

```
module cache
   FIFO req, resp;
   reg  pending;  // a blocking cache
   method cache_req(a) =                          /* step 1 */
      req.enq(a);
      pending := true;
   when (!pending);

   rule hit: when (cachehit(req.first()) =>       /* step 2 */
      resp.enq(cachelookup(req.first());
      req.deq();

   rule miss: when (cachemiss(req.first()) =>     /* step 2 */
      mainmem.req(req.first());
      req.deq();

   rule mainmem_resp: when (true) =>              /* step 3 */
      resp.enq(mainmem.resp());

   method cache_response() =                      /* step 4 */
      resp.deq()
      pending := false;
      return resp.first();
   when (true);
endmodule
```

Figure 5-13: Cache code

We should note that we could have scheduled the *miss* rule to occur separately from the "fast-path". However, the *hit* and *miss* share the result from the cache memory lookup (to determine if a hit occurs). Thus, it does not make architectural sense to schedule the *miss* rule separately from the *hit* rule.

5.7 Generalizations

Several interesting generalizations of the performance guarantee language (ARPG's) and associated scheduling algorithms are possible. Most of these are concerned with how to specify and compile designs that contain multiple performance guarantees.

Thus far, our algorithms simply state that separate composite rules should be generated for each constraint (ARPG). If the rules in one ARPG conflict with the rules in another ARPG, then Hoe and Arvind's scheduler will choose a maximal subset of rules (conditional actions) from each ARPG. For example, suppose rules R_2 and R_3 conflict, ARPG0 is $R_1 < R_2$, and ARPG1 is $R_3 < R_4$. The scheduling algorithms presented in this chapter will ensure that the ARPG's are satisfied, however they do not specify what should happen if for example all four rules' predicates are simultaneously true. Should R_1, R_2 and R_4 execute or should R_1, R_3 and R_4 execute? Either case is valid and satisfies the ARPG's. However, the designer does not have direct control over this scheduling process. We view this as a second order schedule specification problem. However, a richer language to also specify such constraints would be nice.

Another generalization is that each ARPG performance group could contain arbitrary rules or methods. In Section 5.1.3 we stated that each such group should only contain ME of CF rules and methods. The motivation for this restriction was that we did not want to introduce a separate scheduler for each performance group. However, we could allow arbitrary rules in each group and then use Hoe and Arvind's scheduler to choose which rules in each performance group should execute.

5.8 Chapter summary

We have described a method that allows a designer to specify performance constraints by indicating which rules should execute each cycle and in what order they should appear to

execute in. As we have shown, this flow adds flexibility to the design environment by allowing the designer to easily restructure pipelines. We leveraged previous research on design transformation through rule composition to ensure that these transformations do not alter the design's functionality.

We demonstrated the power of the new scheduling algorithms via several examples, most notably a processor pipeline and the FIFO to implement the pipeline stages. We showed that using performance specifications the pipeline can achieve the expected performance. This required the FIFO to be rescheduled such that simultaneous enqueue and dequeue are allowed, and that the value being enqueued can be forwarded to the bypass logic. This was not possible in previous design flows for guarded atomic actions. We also showed that by only changing the performance specification the same processor design could be transformed into a superscalar design or an unpipelined design.

Overall, we believe this flow combines the positive attributes of rule-based design, that is, decoupled specification, with the power the designer expects to ensure correct performance.

Chapter 6

Circuit and Performance Evaluation

The previous chapter introduced a scheduling algorithm that transforms designs to satisfy user-specified performance constraints (ARPG's). This chapter analyzes the circuits that are generated in this compilation flow and presents quantitative results to show that the transformations truly result in the designs we expect—for example that the superscalar processor constraint produces a processor that executes two instructions per cycle.

The chapter contains three main sections. We first examine the FIFO circuit that results from the processor performance constraints and show that it corresponds to precisely the same circuit a designer would have created in a traditional RTL design flow. We then analyze circuit implications for designs that contain multiple performance constraints and introduce a slightly modified EHR to improve performance for such cases. The chapter's last main section analyzes the GCD circuits and processors that result from various performance guarantees. We study the resulting area, cycle time, cycle count, and overall performance for a small benchmark. These results show that many micro-architectures can rapidly be explored by simply changing a design's performance specifications. In addition, we argue that in most cases the cycle time is near optimal. For those cases in which the cycle time does not match our expectations, we show that through small design and circuit generation optimizations a nearly optimal design can be obtained.

6.1 Pipeline FIFO circuits

This section analyzes the FIFO's that are generated during processor synthesis with performance constraints. We argue that a single-element FIFO turns into a single pipeline register with the control structure that a designer would have constructed in a traditional RTL design flow.

Figure 6-1 shows the FIFO circuit that is derived directly from the code in Figure 5-11. The figure shows the EHR structure for the two state elements (*full* and *data*). The interface signals are labeled with both the EHR interface signal names and the corresponding FIFO signals that connect to them. (Note: the bypass^2 output is generated as a combinational function of the *full* and *data* state. All other signals are directly generated from one of the two states.)

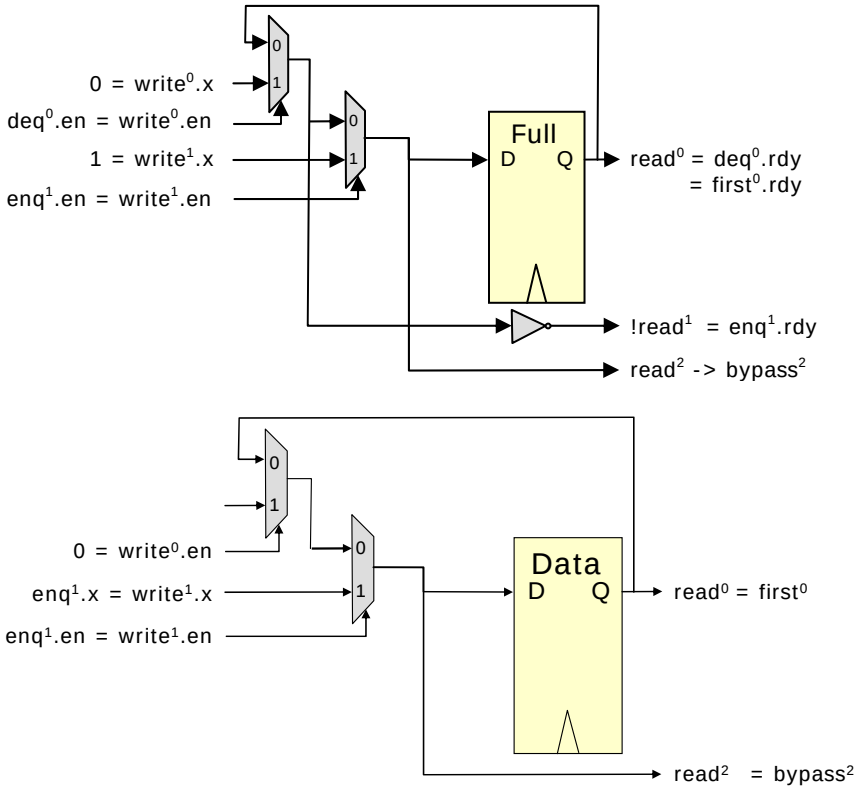


Figure 6-1: Original FIFO circuit

At first this might appear like a lot of logic for a pipeline stage. However, in the next few paragraphs we show that much of the logic disappears after pruning and constant propagation. But first, we highlight several important properties. One important property is

that the $enq^1.rdy$ signal depends on the *full* state and the $deq^0.en$ signal. If $deq^0.en$ is true, then $enq^1.rdy$ will always be true. Hence, if we dequeue from the FIFO, we can always concurrently enqueue into the FIFO. Similarly, the $bypass^2$ outputs depend on the current *full* / *data* states as well as the $deq^0.en$ and $enq^1.en$ signals. If a value is being enqueued into the FIFO, then $bypass^2$ returns the value being enqueued. Otherwise it returns the value already in the FIFO, provided of course a valid value is present (*full* is true). Neither the simultaneous enqueue and dequeue, nor this type of bypass structure could be safely implemented in rule-based design without the rule composition algorithm and the EHR register structure. We should also note that this structure is automatically generated from the performance constraint—the designer does not manually create the mux structures.

The first optimization to the FIFO circuit appears after pruning (see Figure 5-7). The $read^1$ and $write^0$ interfaces are unused in the data registers. Hence they can be pruned, resulting in the circuit shown in Figure 6-2. This data register implementation is optimal—no logic can be removed from it and its structure is equivalent to the data component of a single register pipeline stage. (Note: the feedback from Q to D could instead be implemented as a flip-flop with enable.)

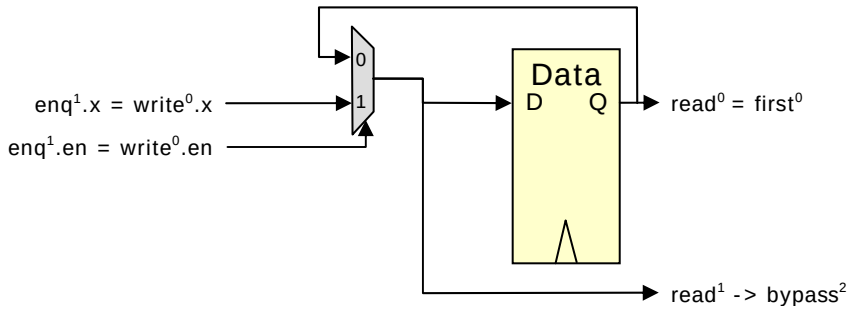


Figure 6-2: Pruned FIFO data register

The full register implementation in Figure 6-1 is optimal if the pipeline is flow controlled. The decode stage is an example of a pipeline stage that exerts upstream flow control since the stage can stall due to a data hazard. During a stall no entry is dequeued from the bF FIFO. Upstream stages must then be flow controlled to prevent overflow of the FIFO that feeds the decode stage (bF). This in turn means that the fetch stage cannot enqueue a new value into the FIFO if the decode stage stalls. Hence, the two stage logic is required at the *full* register input and more importantly, the $enq^1.rdy$ signal depends on the $deq^0.en$ signal. The

dependence of $enq^1.rdy$ on $deq^0.en$ implies that a combinational path is created for upstream pipeline stages—one gate per stage. This is not entirely surprising since flow control must be propagated upstream.

However, not all pipeline stages require flow control. For example, the WB rule will always execute if the bE FIFO contains a value. Hence, for the WB stage the $deq^1.en$ signal is always equal to the $full.read^0$ state. If we propagate this information along with the constant inputs through the *full* register logic we obtain the circuit shown in Figure 6-3. All mux's at the register input are optimized away. Most important though, the $enq^1.rdy$ signal is optimized to always equal 1. This implies that the upstream pipeline stages can always enqueue into a pipeline stage that is not flow controlled. Hence, no combinational control path is created between pipeline stages that always operate synchronously without flow control. (Note: these logic optimizations are automatically performed during logic synthesis using a gate-level synthesis tool such as Synopsys Design Compiler.)

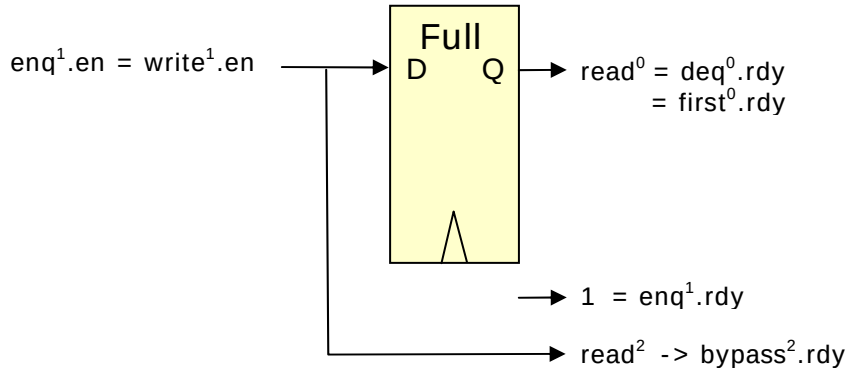


Figure 6-3: No flow control full register (deq.en = deq.rdy)

In summary, the single-element FIFO implementation using EHR's produces exactly the circuits that a designer expects. Hand-coded single register pipeline stages will not perform better as back pressure logic is created only when required.

As a comparison we show the FIFO for a "flow-through" design in Figure 6-4. This style FIFO is created if we reverse the order of the standard processor pipeline performance constraints to:

```
F < D < E < WB
```

This constraint says to execute the fetch rule, then the decode rule, etc.—all within one cycle. Hence, we expect instructions to flow through the pipeline and not be registered (we have transformed the design into an unpipelined design.).

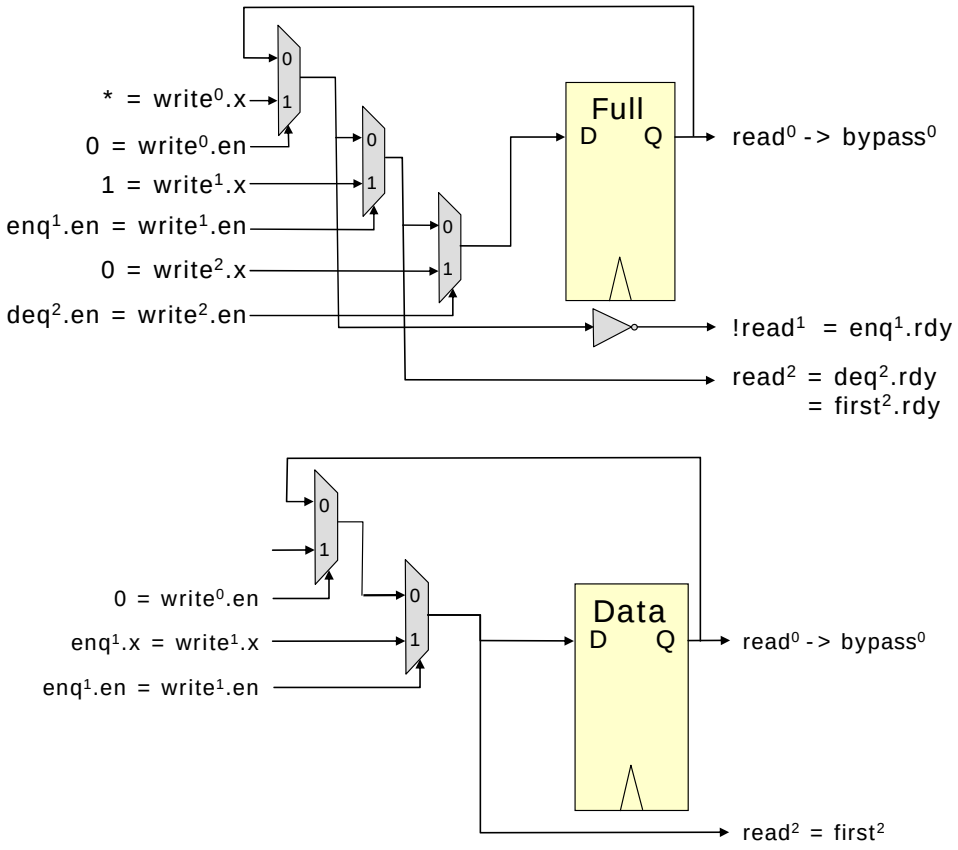


Figure 6-4: Flow-through FIFO circuit

As expected from the performance constraint, this FIFO does not latch a value if *enq* and *deq* are simultaneously enabled—*enq* sets *full* to 1, and within the same cycle, but appearing to occur afterwards, *deq* sets *full* to 0.

Similar to the standard pipeline FIFO, these circuits are optimized if we perform pruning, constant propagation, and the compiler can detect for example that the *deq* method will always be enabled when ready. We show the circuit after optimization in Figure 6-5.

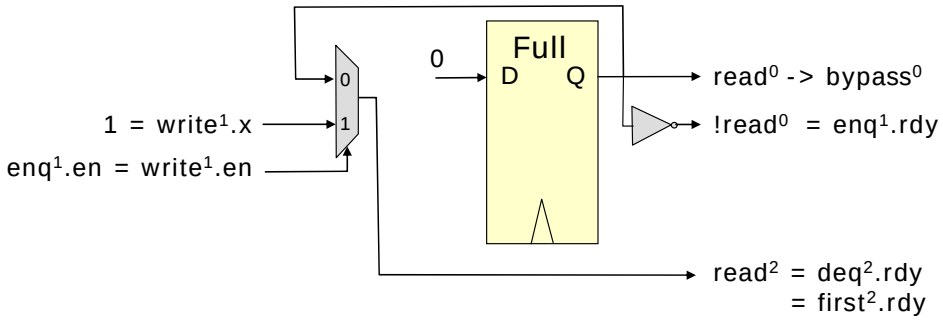


Figure 6-5: Flow-through FIFO circuit optimized (1)

This circuit can be further optimized if we allow constant registers to be eliminated (most gate-level synthesis flows allow such an optimization). The resulting circuit is shown in Figure 6-6. Similar optimizations occur on the data register.

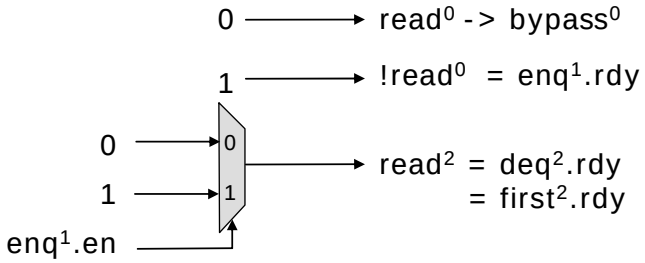


Figure 6-6: Flow-through FIFO circuit optimized (2)

6.2 Multi-constrained modular composition

Our performance driven synthesis algorithm produces correct implementations when multiple ARPG constraints are specified. However, the resulting circuits can introduce critical paths that the designer might not have intended. We can illustrate this problem via the processor example from the previous chapter. A natural constraint for the 4-stage processor is:

```
WB < {E_jz_nottaken, E_Add} < D < F
WB < E_jz_taken
```

This constraint states that the non branch taken rules should appear to execute in the usual order. Since the branch taken rule should only execute with the write back rule (the other stages are cleared), this part of the constraint is written into a separate ARPG.

If we call the PERFORMANCESCHEDULE procedure with these two constraints as input we obtain the following indexed methods for the *pc* register:

```
write[1]  - due to the E_jz_taken rule
read[3]   - due to the F rule
write[3]  - due to the F rule
```

An example of an unintended combinational path is the *pc* value produced by the E_jz_taken rule ($write^1$) being forwarded to the F rule ($read^3$). Although functionally correct, this circuit is likely to produce a design with unsatisfactory cycle time. The forwarding path might never be used because a scheduler disallows the E_jz_taken and F rules from executing concurrently, but the path does exist and hence would be considered as a real timing path during gate level synthesis. One option is to mark such paths as *false paths*. Another option is to rely on an automated false path detection tool. However, false paths generally complicate the gate-level synthesis and physical design process. Thus, we introduce a slight modification on the EHR circuit to avoid this problem.

Our solution is based on a notion of "separate" EHR interface groups for each ARPG. Each interface group allows values to be forwarded among the read and write methods, as is the case in the conventional EHR. However, we do not allow values to be forwarded from one group to another. In the processor example above, such grouping results in the following *pc* interface method calls. Here we call the first ARPG group *a* and the second ARPG group *b*:

```
write[1b]   - due to the E_jz_taken rule
read[3a]    - due to the F rule
write[3a]   - due to the F rule
```

To complete this idea, we present a diagram of the split EHR in Figure 6-7. The priority mux in this circuit is driven by the write enable inputs and gives preference to any writes on the *b* interfaces. The scheduling constraints for this register are:

$$read^{a0} < write^{a0} < read^{a1} < write^{a1} < read^{a2} < write^{a2}$$

$$read^{b0} < write^{b0} < read^{b1} < write^{b1} < read^{b2} < write^{b2}$$

$$\{read^{a0}, write^{a0}, read^{a1}, write^{a1}, read^{a2}, write^{a2}\} < \{write^{b0}, write^{b1}, write^{b2}\}$$

(Note: values are only forwarded from $write^{a*}$ to $read^{a*}$ and from $write^{b*}$ to $read^{b*}$, and not from $write^{a*}$ to $read^{b*}$ or $write^{b*}$ to $read^{a*}$. It should be clear that as with the EHR, this "split" structure can be generated for arbitrary conditional method interfaces.)

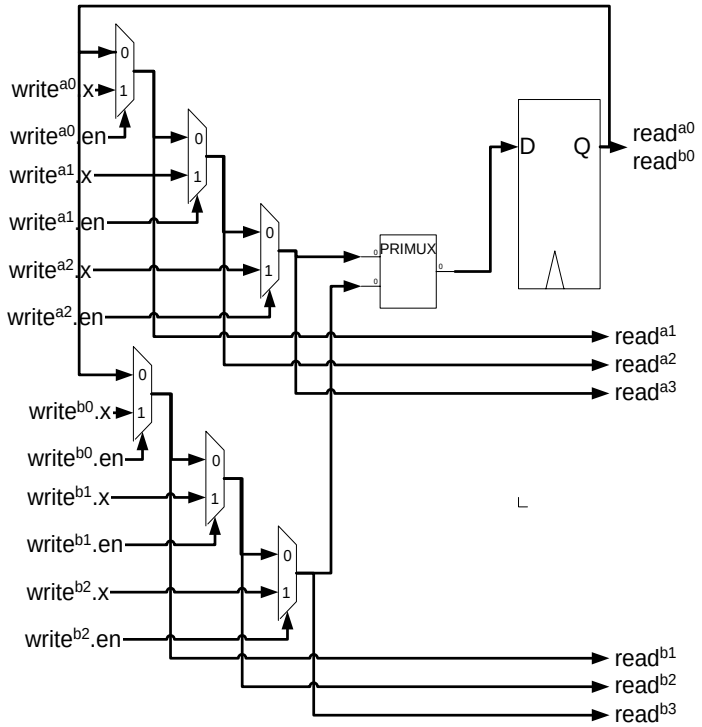


Figure 6-7: Split EHR

6.3 Processor and GCD evaluation

We evaluated the new synthesis methodology to confirm that it produces functionally correct results, that the performance meets the designer's expectations, and that the final circuit quality remains high. To implement the new flow, we created the EHR state-element in Verilog and imported it, along with its interface scheduling properties, into Bluespec. We then created the designs using registers as the only primitive state elements, that is, FIFO's, RF's, etc. were created in Bluespec from registers only. We then transformed the design into a new design according to the procedure outlined in Chapter 5 and Section 6.2 for each scheduling requirement. The resulting design was then fed through the Bluespec compiler to produce RTL Verilog, which was then synthesized using Synopsys Design Compiler to generate area and timing numbers for the TSMC 0.13µm G process. We generated area and timing numbers for two different timing constraints to illustrate numbers for both an area and a timing-constrained synthesis run. We also simulated each design to measure functional performance.

The design transformations in the initial run of these experiments were performed by hand. We have since developed in C/C++ an automated compilation step which accepts a design specified in a subset of the Bluespec language along with a performance specification as input. As output, it generates a transformed Bluespec design along with any EHR circuits that the design requires. We have successfully applied this automated process to the processor designs as well as several other small examples.

Figure 6-8 shows the results for GCD designs (see Figure 2-2) with 3 different scheduling constraints. The first design is the original design and does not incorporate any transformations. The second design composed $R_{swap} < R_{sub}$, and the third design was scheduled to satisfy the constraint: $R_{swap} < R_{sub} < R_{swap} < R_{sub}$. As is expected, as more rules are composed, fewer cycles are required to compute results. Similarly, the critical path increases as more rules are composed. In spite of this, for the 5ns constrained synthesis, the two constrained schedules speedup the GCD execution by 1.06 and 1.98. However, the area of the two constrained designs increases by 52% and 350% over the baseline unconstrained design. The area increase may come as a surprise because the 4-way composed rule should not be using 4 times the area since registers are not replicated and we only increase the number of adders from one to two. However, because the 4-way composed design is unable to make the timing constraint of 5ns, its adders are substantially larger than those in the other two designs so as to improve its timing. (Note: the critical path in the 4-way composed design is 32-bit compare (R_{swap}) followed by 32-bit compare and 32-bit add (R_{sub}), followed by 32-bit compare (R_{swap}) followed by 32-bit compare and 32-bit add (R_{sub}).)

GCD Input	Measure	No Constr.	$R_{swap} < R_{sub}$	$R_{swap} < R_{sub} < R_{swap} < R_{sub}$
Input 1	Cycles	91	78	39
Input 2	Cycles	117	101	51
10ns constr.	Area (μm^2)	5221	6479	13705
10ns constr.	Timing (ns)	10	10	10
5ns constr.	Area (μm^2)	5909	9003	26638
5ns constr.	Timing (ns)	4.54	5.00	5.3
5ns exe. time	ns	472	448	239
5ns speedup		1	1.06	1.98
5ns area inc.		0%	52%	351%

Figure 6-8: GCD results

This GCD example may appear trivial, however we were able to generate these numbers simply by changing the performance constraints and then running the same design through the tool chain. Even for such a simple example, the effort to manually code each case in RTL would take more effort to first write and then verify.

Next, we look at a more complex example. Figure 6-9 shows the results for a 4-stage processor pipeline. This processor has the same pipeline structure as that discussed in Figure 5-9 and Figure 5-8. The major difference is that we have added more instructions so that we could run simple programs. In addition to an unconstrained design (the traditional Bluespec flow), we synthesized the designs with the following four schedules, where E* = contains all the execute rules except for the jump taken rule (E_jz_taken):

```
Schedule1:    WB  < E  < D  < F

Schedule2:    WB < E* < D  < F
              WB < E_jz_taken

Schedule3:    WB < E* < D < F < E_jz_taken

Schedule4:    WB < WB < E* < E* < D < D  <  F < F
              WB < WB < E_jz_taken <  E_jz_taken
```

We had discussed the rational for the ordering in Schedule 1 in the previous chapter. However, architects usually optimize the branch-taken case differently from the branch-not-taken case and this is what is reflected in Schedules 2 and 3. In Schedule 2 we exclude the branch-taken rule from the first performance specification expecting to make the critical path shorter than Schedule 1 because fetch now cannot observe the branch target in the cycle that the branch is resolved. This effectively splits the access to the PC between the fetch and branch resolution stage (see Section 6.2). This eliminates the critical path from Schedule 1 but in turn should have a slightly higher cycle count since branch taken and fetch cannot execute in the same cycle. In Schedule 3 we move the branch taken rule to the "end of the cycle". This eliminates the critical path from branch-taken through fetch. However, this means that the branch taken observes the results of the decode stage—effectively we have moved the branch resolution into the decode stage. Hence the critical path becomes: execute an add instruction, bypass it into the decode stage and compare it with 0 to see if the branch is taken. This is a long critical path, but is a design used in many processors. Finally Schedule 4 is the 2-way superscalar version of Schedule 2.

Design	Bench. (cycles)	Area 10ns (μm^2)	Timing 10ns (ns)	Area 1ns (μm^2)	Timing 1ns (ns)	Exec. Time (ns)	Speedup	Area over-head
1 element fifo:								
No Constr.	18525	24762	5.8	33073	1.6	29640	1.00	0%
Schedule1	9881	25362	7.5	34161	2.2	21738	1.36	3%
Schedule2	11115	25001	6.6	34511	1.9	21119	1.40	4%
Schedule3	9881	25180	8.0	34896	2.6	25691	1.15	6%
Schedule4	11115	25264	6.8	36037	1.9	21119	1.40	9%
2 element fifo:								
No Constr.	18525	32240	7.4	39033	1.9	35198	0.84	18%
Schedule2	11115	32535	8.4	47084	2.63	29232	1.01	42%
Schedule4	7410	45296	10.0	62649	4.7	34827	0.85	89%
Schedule4+Fix	7410	40180	9.9	62053	3.0	22230	1.33	88%

Figure 6-9: 4-stage processor results

We synthesized the designs using one and two-element FIFO's as pipeline stages since a two-element FIFO is required for a superscalar implementation to perform well. A simple benchmark loop with arithmetic operations and conditional branches was run on all designs. Although this benchmark was very small, it provides an idea of the relative throughput for each processor pipeline. The execution time can be computed by multiplying the cycle time and the cycle count. We compute the speedups and overhead by treating the unconstrained Bluespec schedule with 1-ns timing constrained synthesis and a single-element FIFO as the base case.

Component	Propagation Delay
32 bit addition	0.9ns
32 bit increment	0.6ns
32 bit compare to 0	0.6ns
2-1 mux (32 bits wide)	0.3ns
Clk to Output + Setup Time	0.4ns

Figure 6-10: Component delays

As a reference for the timing results, we show timing numbers for some of the key processor components in Figure 6-10. These numbers are approximate since each synthesis run

selects slightly different implementations. However, it is clear that unless we further pipeline the design, no design can have a cycle time of much less than 1.6ns since we must sequentially get the decode FIFO output (Clk to Q—about 0.3ns), pass through an adder in the execute stage (about 0.9ns), pass through at least one level mux (0.3ns) and then satisfy setup time (0.1ns).

As expected, Schedule 1's total execution time is much better than the unconstrained implementation because the standard Bluespec compiler can only schedule alternating stages to execute in each cycle. It shows a speedup of 1.36 with only a 3% increase in the area. Schedule 2 improves on this by showing a speed up of 1.40 with a 4% increase in the area. We really did not expect an improvement with Schedule 4 with one-element FIFO's since a superscalar design will only function with better throughput if two-element FIFO's are used! The area also did not increase for the superscalar one-element FIFO case because the duplicate rules are optimized away in the compilation phase (Synopsys Design Compiler recognizes that the logic is never used). We should note that as in the GCD example, these experiments could be performed just by changing the scheduling specifications; the algorithms we presented earlier ensure that the correctness of the designs is maintained in this process and that the designs are transformed to satisfy the scheduling requirements.

The results for two-element FIFO's in Figure 6-9 show the cycle count improvements for the superscalar design but also significantly worse clock speeds. The speedup in the best case is only 1.33. This is partially due to the penalty of clearing the pipeline after each branch taken is relatively high in the superscalar design. However, somewhat disturbingly, the cycle time for the superscalar design is more than twice that of the single-element FIFO composed design (4.7ns vs. 1.9ns). In an optimal implementation we would expect the superscalar design to have a cycle time of only slightly more than the composed pipeline (about two additional mux stages, or about 0.6ns). Below we discuss several simple changes we can make to the circuit generation and the FIFO implementation to reduce the superscalar cycle time from 4.6ns to 3.0ns (about 0.5ns within optimal). (Note: This is the only design for which we altered code to improve cycle times—all other designs were directly derived from the original processor code and transformed using the conditional composition algorithms.)

The first change is a circuit transformation shown in Figure 6-11. This is a simple logic transformation that Synopsys Design Compiler currently does not perform, but which is easy to add to the Bluespec compilation. In this case, the *Bz1_taken* signal is on the critical path. In the original design (on the left side of the figure) the next PC computation for the second fetch

in the superscalar fetch stage cannot be computed until the earlier branch is resolved. By simply moving logic across the mux we can improve this path.

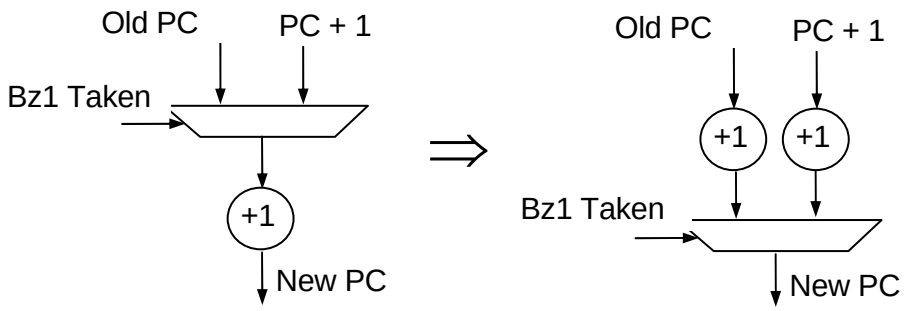


Figure 6-11: Moving logic across a mux

A more interesting change that had a dramatic impact on the cycle time of the superscalar design is that we slightly changed the two-element FIFO specification. These changes do not alter the behavior of the FIFO, but embed high-level knowledge that we have about the FIFO into its circuits. For example, we know that after dequeueing from the FIFO twice, it will be empty. Since the write back stage in the superscalar design will always execute twice if the FIFO contains two valid elements (and once if it contains only one element), the execute stage does not need to check that the FIFO between the write back and execute stages is empty. Such a check can add one or two mux's to the critical path (0.6ns). We can achieve this effect by rewriting the *enq* method as follows (the changes to this method are highlighted in italics):

```
enq(x)    =     if  (full0 == 0) then
                   data0 := x;
                   full0 := 1;
                   full1 := 0
                else
                   data1 := x;
                   full1 := 1;
     when ((full1 == 0)  || (full0 == 0))
```

Clearly, these changes do not alter the behavior of the design: We know that if *full0* is 0, then *full1* is also always 0, so it is safe to add the check of *(full0 == 0)* to the method's implicit condition. Similarly, we can write the value 0 to *full1* if the FIFO is empty and we are enqueueing a value since the value will be placed in the "0" slot. Although these changes do not change the functionality, they have the impact of allowing constants to be effectively propagated through the pipeline—for example after this change, the execute stage logic is

optimized via constant propagation to no longer need to check if the FIFO it is enqueueing in is full. (In Section 6.1 we showed that such an optimization automatically happens in the single-element FIFO case. For the two-element FIFO the above changes are required to make the constants propagate effectively.)

Another example of this type of change is to the FIFO *clear* method. Again we highlight the change in italics. Obviously, the data values can have any value after the FIFO is cleared. However, by setting the *data0* value during a *clear* method call to the value it would have after a *deq*, the logic that reads from the FIFO can be optimize: regardless of what the "first" rule in a stage does (*deq* or *clear*), it always moves *data1* into *data0*, so the "second" rule to execute always knows what the "new" value in *data0* will be and hence can directly look at the *data1* register. We illustrate these cases in Figure 6-12. Again, by simply adding this line of code which clearly does not change FIFO functionality we embed some high-level knowledge into the design. The result is that a mux stage for one of the FIFO's is removed from the critical path. (Note: this optimization works in the processor execute stage where the "first" execute rule always executes. However, this optimization does not improve timing for the decode rules because the "first" decode rule might stall.)

```
clear' =   full0 := 0;
           full1 := 0;
           data0 := data1;
     when (true)
```

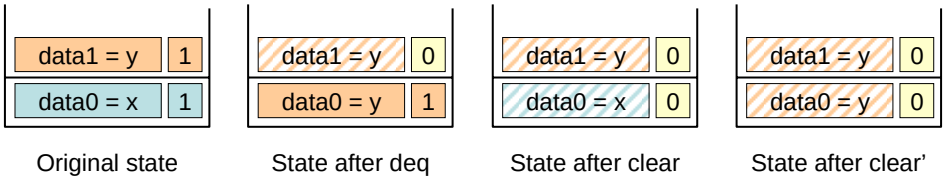


Figure 6-12: FIFO states after deq and clear operations

These types of changes allowed us to reduce the cycle time from 4.6 to 3.0ns. The remaining 0.5ns can be obtained through similar changes but they become counterintuitive since one needs to keep track of when data is available and how mux's are introduced. Instead, at that point it would be more reasonable to rewrite the design as a superscalar design. It is important to recognize that a decision to rewrite the design with "superscalar" in mind is not due to a short-coming in the synthesis methodology that we present here. As designers we simply have high-level knowledge that the compiler does not have. Without this knowledge,

the compiler must be conservative. An interesting future approach to this work might be to use user-assertions to guide the compilation process. For example, an assertion could be added that if FIFO slot "0" is empty, then FIFO slot "1" is also empty. Such assertions would ensure that the logic is optimized more efficiently.

6.4 Chapter summary

This chapter presented quantitative data that shows that the performance driven synthesis algorithm works correctly, and in many cases as efficiently as a designer would expect. We showed that several reasonable processor micro-architectures could be generated by simply changing the performance constraints. We were somewhat troubled by the dramatic increase in cycle time for the superscalar design. However, after carefully reviewing the critical paths we were able to implement specific optimizations for the two-element FIFO which moved the superscalar design closer to its optimal implementation.

Chapter 7

Related Work

High level design specification and synthesis is an active area of academic research and industry development. This chapter reviews some of the work others have performed in this area. We discuss related work on hardware specification and synthesis using guarded atomic actions, "traditional" behavioral synthesis, synchronous languages, and processor based synthesis. The goal in all these approaches is to allow designers to more effectively take advantage of the tremendous resources that are available in state-of-the-art semiconductors.

7.1 Guarded atomic actions

The idea of using guarded atomic actions to describe distributed systems was developed many years ago[10, 15, 26, 33, 36, 39] and popularized in[10] via the UNITY programming language. More recently, guarded atomic have been used in the hardware domain. Initial successes arose in the area of hardware verification, for example Dill's Murphi[16] system allowed cache coherence protocols to be verified. Initial work on hardware design specification and synthesis using guarded atomic actions was performed in Staunstrup's Synchronous Transactions[51], Sere's Action Systems[43], and Arvind and Shen's TRS's[3]. These systems used basic processor pipelines to demonstrate the practicality of their

approaches. However, Arvind and Shen's research focused on more complex designs such as reorder buffers[3] and cache coherence protocols[50, 52]. Staunstrup also demonstrated synthesis capabilities but the amount of concurrency he was able to derive among rules was so limited to make his system impractical for realistic hardware design.

Hoe and Arvind were the first to show that sufficient parallelism can be found among rules to make hardware synthesis from guarded atomic actions practical[27-29]. They assumed a flat design environment in which each rule can interact with registers, FIFO's, and register files. Using such a system they demonstrated that many designs can be efficiently implemented using guarded atomic actions. Their work constitutes the foundation for much of research presented in this thesis.

More recently, larger scale design exploration and more sophisticated synthesis systems have been introduced. More advanced processors have been synthesized and simulated[11-13, 48, 57], and current effort's in Arvind's group are underway to develop a full-blown PowerPC simulation and synthesis environment. A dramatic advance in synthesis robustness is due to the commercial development of the Bluespec language and synthesis tool[4, 8]. Interesting research is also being conducted to merge ideas from synchronous languages with Bluespec[40], and to make assertions a core part of the Bluespec synthesis environment[42]. Related work on pipeline transformations in a system of guarded atomic actions appears in[37, 38]

7.2 Traditional behavioral synthesis

Traditional behavioral hardware synthesis is based on control-data flow graphs (CDFG's), and many projects have successfully transformed and optimized CDFG's into circuits[18, 23, 30, 32, 44]. The major difference between the CDFG synthesis flows and synthesis from atomic actions is that CDFG's focus on generating an efficient *static* schedule of operations over a sequence of control steps. In contrast, rule-based synthesis generates a scheduler that *dynamically* determines which rules fire in every cycle. We believe dynamic scheduling is important in hardware systems because many designs have (i) a large number of data dependent conditional paths, each with its own timing and resource requirements, (2) have subsystems with variable and unpredictable latencies (due to caching and interference from other processes, etc.), and (iii) have input events whose timing is often unpredictable. We

believe static schedules produce good results for many DSP type applications but are not well suited for more complicated micro-architectures that combine data-paths with complex control logic.

Although the motivation is slightly different, we should point out that some of the ideas in the EHR logic overlap with ideas in CDFG synthesis. For example, chaining is presented in[18] as a mechanism to improve performance by forwarding the value from one operation to another without storing an intermediate result. Dynamic renaming is used in[23] to eliminate data dependencies that limit code motion, and hence allows more aggressive compiler optimizations to be implemented.

7.3 Other efforts

Synchronous specification languages are another active area of research in hardware specification. Examples are Esterel[7, 17], Signal[20], and Lustre[9] which were all designed to deal with real-time issues[6]. Berry[7] and Edwards[17] have presented methods to generate hardware from Esterel but these efforts have yet to yield high quality hardware in comparison to synthesis from Verilog RTL.

Another type of research has focused on synthesis of specialized versions of programmable processors[1, 24, 49, 56]. These efforts are only tangentially related to general purpose HDLs because the primary focus is on processor issues such as instruction encodings and the automatic generation of assemblers, compilers, etc. Several companies, most notably Tensilica Inc, and more recently Stretch Inc., have shown that a market for such products exists. However, many applications continue to require the performance, power, and cost benefits of the RTL / gate-level solutions we address in this thesis.

Many other projects on high-level synthesis have been worked on. Relevant to the work in this thesis is the Liberty micro-architectural exploration tool[55], SystemC[21, 41, 53], and the Scenic design system[34].

Chapter 8

Summary and Future Work

This thesis presented new synthesis algorithms and design specification constructs that enhance the designer's ability to easily express complex architectures. The two main contributions are (i) a modular synthesis flow that adds semantics to module interfaces, and (ii) a performance driven synthesis algorithm that allows a designer to specify what portion of a design should execute concurrently in each cycle and what order these components should appear to execute in. Both of these contributions have immediate practical benefits in the context of rule-based design because they substantially improve synthesis times and allow a designer to more easily ensure that sufficient parallelism is achieved among a design's rules. In addition, we hope that the thesis leads to an enhanced design flow in which designers attempt more aggressive architectures and experiment with micro-architectural alternatives rather than choose conservative and often wasteful implementations as is all too common these days.

To this end, the modular synthesis flow makes design exploration and re-use easier than in traditional hardware design by incorporating interface scheduling annotations into the design specification. These annotations can be compiler-derived or manually inserted, and indicate how the logic that connects to the module must be scheduled. This eliminates the error prone process of reading through informal design specifications and searching for the interface use restrictions. The modular synthesis flow also eliminates the tedium of manual coding of the scheduling logic that glues logic and modules together. This allows modules to be swapped

in and out of a design, for example to evaluate performance / timing tradeoffs, without having to worry about the connecting logic.

The performance driven synthesis flow ensures that a designer achieves the parallelism and throughput that are expected. It also allows designers to easily experiment with micro-architectural alternatives without changing the underlying rules—only the performance specifications need to change for many of these experiments. We demonstrated several such examples via a processor pipeline that could be transformed from a pipelined design into an unpipelined design, a superscalar design, or a pipelined design with alternate branch resolution logic, simply by changing a one-line performance specification.

In summary, we believe modular rule-based design with performance specification is an attractive model for design specification and synthesis. We hope this design style is adopted and leads to interesting, complex, and higher performing designs than is possible with traditional design methodologies.

8.1 Future work

The two main topics of this thesis that could be further investigated are modular synthesis of non-tree module call hierarchies and the quality of circuits generated during performance driven synthesis. Both areas present interesting research problems and solving them would have immediate practical benefits for the designer.

The reason that modular synthesis of non-tree module call hierarchies is an important problem to solve is that many designers naturally create non-tree structures. The flow described in this thesis requires that such hierarchies must be transformed into a tree-structure via selective merging of modules before modular compilation can be performed. Although this is an automated process, we do not achieve the full benefits of modular compilation if some modules have to be merged for the synthesis algorithms to be applicable. An improved modular compilation flow would accept designs with arbitrary hierarchies and generate logic for each module individually. Such a flow is likely to require logic in between modules or global scheduling logic / knowledge—something we were able to avoid in the synthesis flow for tree hierarchies.

The second important area that deserves further research is the quality of the circuits generated in the performance driven synthesis flow. As we showed, many of the generated

circuits match the circuit quality that optimized hand-generated RTL would produce. However, as the superscalar transformation illustrated, some constraints lead to sub-optimal circuits. We showed that through code and circuit modifications the results for the superscalar designs could achieve nearly optimal performance. However, this was a somewhat cumbersome process and an automated approach would be preferable.

An automated approach to improving the circuit quality of the performance driven synthesis results is likely to contain two components. One component simply improves the gate-level synthesis of circuits. These changes preserve the functionality (next-state values) of the original design. An example of such a transformation was shown in Figure 6-11. We believe a small set of such transformations will solve many of the circuit generation issues. If critical path feedback can be generated during the compilation from rules into RTL, such transformations could be inserted at the RTL level. Otherwise they can be added during the RTL to gates synthesis step.

The second approach that improves circuit quality is incorporating a designer's high-level knowledge about the design. Allowing a designer to express such knowledge in the form of assertions and then using these assertions for improved synthesis is an attractive proposition. Formal verification tools could attempt to prove the assertions and new synthesis algorithms would use the assertions to generate more efficient circuits. We view this as a challenging problem but one to likely lead to an improved design experience.

In summary, several approaches for an automated solution to attacking the circuit quality problems are possible. Although many of the circuits we generate are optimal (or close to it), we believe this is a very interesting area for future research. The results from such work are also likely to be applicable to many other high-level synthesis flows where similar problems are encountered.

Next, we point out two additional areas for future research. We do not view these as important as the previous two, but they present interesting problems and their solution would improve the design flow. The first area is to extend the scheduling annotations to incorporate additional information. Examples of possible extensions are parameter dependent annotations (for example, if the inputs to two methods are 0, then their annotation is "<", otherwise "C"), or annotations that are dependent on the state of the system (for example, a FIFO which has the $enq < deq$ property if the FIFO is empty, but otherwise satisfies $deq < enq$). These extensions require a more dynamic scheduler since annotations are no-longer fixed. However, they allow a designer to create more flexible designs.

Finally, an interesting topic for future research is extending the performance specification language. Rather than view the performance specification as a set of linear constraints, we could imagine a language that allows arbitrary (non-cyclic) constraint graphs. Such constraints are likely to make it easier to specify the desired performance for larger designs. Clearly, the performance driven synthesis algorithms would have to be modified to support such constraints but we believe the basic synthesis ideas (the EHR and the numbering of rules and methods) would remain.

Bibliography

1. Aditya, S., Ramakrishna Rau, B. and Kathail, V., Automatic architectural synthesis of VLIW and EPIC processors. in *12th International Symposium on System Synthesis*, (1999), 107-113.
2. Arvind, Nikhil, R.S., Rosenband, D.L. and Dave, N., High-level Synthesis: An Essential Ingredient for Designing Complex ASICs. in *ICCAD*, (San Jose, 2004).
3. Arvind and Shen, X. Using term rewriting systems to design and verify processors. *Micro, IEEE*, *19* (3). 36-46.
4. Augustsson, L. and others. Bluespec: Language definition, Sandburst Corp., 2001.
5. Baader, F. and Nipkow, T. *Term Rewriting and All That*. Cambridge University Press, Cambridge, UK, 1998.
6. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P. and de Simone, R. The synchronous languages 12 years later. *Proceedings of the IEEE*, *91* (1). 64-83.
7. Berry, G. Esterel on hardware. *Philos. Trans. Roy. Soc. London* (Series A, 339). 87-104.
8. Bluespec, I. *Bluespec System Verilog V3.8 Reference Guide*, 200 West St, Suite 402, Waltham, MA 02451, 2005.
9. Caspi, P., Pilaud, D., Halbwachs, N. and Plaice, J.A., LUSTRE: a declarative language for real-time programming. in *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, (Munich, West Germany, 1987).
10. Chandy, K.M. and Misra, J. *Parallel program design : a foundation*. Addison-Wesley Pub. Co., Reading, Mass., 1988.
11. Dave, N. Designing a Processor in Bluespec *EECS*, Massachusetts Institute of Technology, Cambridge, 2004.
12. Dave, N., Designing a Reorder Buffer in Bluespec. in *MEMOCODE*, (San Diego, 2004).

13. Dave, N., Ng, M.C. and Arvind, Automatic Synthesis of Cache-Coherence Protocol Processors Using Bluespec. in *MEMOCODE*, (Italy, 2005).

14. Dave, N. and Pellauer, M., UNUM: A General Microprocessor Framework Using Guarded. in *Workshop on Architecture Research using FPGA Platforms (WARFP)*, (San Francisco, CA, Jan. 2005).

15. Dijkstra, E.W. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, *18* (8). 453-457.

16. Dill, D.L. The Murphi verification system. in *Proceedings of the Eigth International Conference on Computer-Aided Verification*, Springer-Verlag, 1996.

17. Edwards, S.A., High-level Synthesis from the Synchronous Language Esterel. in *Proceedings of the International Workshop of Logic and Synthesis (IWLS)*, (New Orleans, Louisiana, 2002).

18. Gajski, D.D. *High-level synthesis : introduction to chip and system design*. Kluwer Academic, Boston, 1992.

19. Gajski, D.D. *SpecC : specification language and methodology*. Kluwer Academic Publishers, Boston, 2000.

20. Gautier, T., Guernic, P.L. and Besnard, L., SIGNAL: A declarative language for synchronous programming of real-time systems. in *Proc. of a conference on Functional programming languages and computer architecture*, (Portland, Oregon, 1987).

21. Grotker, T. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, 2002.

22. Gupta, P., Lin, S. and McKeown, N., Routing lookups in hardware at memory access speeds. in *INFOCOM '98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, (1998), 1240-1247 vol.1243.

23. Gupta, S., Dutt, N., Gupta, R. and Nicolau, A., SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. in *VLSI Design, 2003. Proceedings. 16th International Conference on*, (2003), 461-466.

24. Hadjiyiannis, G., Hanono, S. and Devadas, S., ISDL: An Instruction Set Description Language For Retargetability. in *Proceedings of the 34th Design Automation Conference (DAC)*, (1997), 299-302.

25. Hennessy, J.L. and Patterson, D.A. *Computer Architecture a Quantitative Approach*. Morgan Kaufman, 1996.

26. Hoare, C.A.R. *Communicating sequential processes*. Prentice/Hall International, Englewood Cliffs, N.J., 1985.

27. Hoe, J.C. Operation-centric hardware description and synthesis *Dept. of Electrical Engineering and Computer Science*, Massachusetts Institute of Technology, 2000, 139 p.

28. Hoe, J.C. and Arvind Operation-centric hardware description and synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, *23* (9). 1277-1288.

29. Hoe, J.C. and Arvind, Synthesis of operation-centric hardware descriptions. in *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, (2000), 511-518.

30. Hwang, C.-T., Lee, J.-H. and Hsu, Y.-C. A formal approach to the scheduling problem in high level synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, *10* (4). 464-475.

31. Kane, G. and Heinrich, J. *MIPS RISC Architectures*. Prentics-Hall, Inc., 1992.

32. Knapp, D., Ly, T., MacMillen, D. and Miller, R. Behavioral synthesis methodology for HDL-based specification and validation in *Proceedings of the 32nd ACM/IEEE conference on Design automation* ACM Press, San Francisco, California, United States 1995 286-291

33. Lamport, L. Specifying Concurrent Program Modules. *ACM Trans. Program. Lang. Syst.*, *5* (2). 190-222.

34. Liao, S., Tjiang, S. and Gupta, R., An Efficient Implementation Of Reactivity For Modeling Hardware In The Scenic Design Environment. in *Proceedings of the 34th Design Automation Conference (DAC)*, (1997), 70-75.

35. Lis, M.N. Superscalar Processors via Automatic Mircoarchitecture Transformations, Massachusetts Institute of Technology, Cambridge, MA, 2000.

36. Lynch, N.A. *Atomic transactions*. Morgan Kaufmann Publishers, San Mateo, Calif., 1994.

37. Marinescu, M.-C. and Rinard, M., High-level specification and efficient implementation of pipelined circuits. in *Design Automation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific*, (2001), 655-661.

38. Marinescu, M.-C.V. and Rinard, M., High-level automatic pipelining for sequential circuits. in *System Synthesis, 2001. Proceedings. The 14th International Symposium on*, (2001), 215-220.

39. Milner, R. *A calculus of communicating systems*. Springer-Verlag, Berlin ; New York, 1980.

40. Nordin, G. and Hoe, J.C., Synchronous Extensions to Operation-Centric Hardware Description Languages. in *MEMOCODE*, (San Diego, June, 2004).

41. Panda, P.R., SystemC - a modeling platform supporting multiple design abstractions. in *System Synthesis, 2001. Proceedings. The 14th International Symposium on*, (2001), 75-80.

42. Pellauer, M., Lis, M., Baltus, D. and Nikhil, R., Synthesis of Synchronous Assertions with Guarded Atomic Actions. in *MEMOCODE*, (Verona, Italy, July, 2005).

43. Plosila, J. and Sere, K., Action systems in pipelined processor design. in *Proceedings Third International Symposium on Advanced Research in Asynchronous Circuits and Systems*, (1997), 156-166.

44. Raghunathan, A. and Jha, N.K., Behavioral synthesis for low power. in *Computer Design: VLSI in Computers and Processors, 1994. ICCD '94. Proceedings., IEEE International Conference on*, (1994), 318-322.

45. Rosenband, D.L., The Ephemeral History Register: Flexible Scheduling for Rule-Based Designs. in *MEMOCODE*, (2004).

46. Rosenband, D.L. and Arvind, Hardware Synthesis from Guarded Atomic Actions with Performance Specifications. in *ICCAD*, (San Jose, 2005).

47. Rosenband, D.L. and Arvind, Modular Scheduling of Guarded Atomic Actions. in *Proceedings of the 41st Design Automation Conference (DAC)*, (2004).

48. Roy, J., High-level Modeling and FPGA Prototyping of Microprocessors. in *Internation Symposium on Field Programmable Gate Arrays (FPGA)*, (February, 2003).

49. Schliebusch, O., Hoffmann, A., Nohl, A., Braun, G. and Meyr, H., Architecture implementation using the machine description language LISA. in *Proceedings 7th Asia and South Pacific Design Automation Conference (ASP-DAC)*, (2002), 239-244.

50. Shen, X. Design and verification of adaptive cache coherence protocols *Dept. of Electrical Engineering and Computer Science*, Massachusetts Institute of Technology, 2000, 178 p.

51. Staunstrup, J. and Greenstreet, M.R. From High-Level Descriptions to VLSI Circuits. *BIT*, *28* (3). 620-638.

52. Stoy, J., Shen, X. and Arvind. Proofs of Correctness of Cache-Coherence Protocols. in Oliveira, J.N. and Zave, P. eds. *Formal Methods Europe*, Springer-Verlag, 2001, 43-71.

53. SystemC Language Working Group. Functional specification for SystemC 2.0.1, 2002.

54. Terese *Term Rewriting Systems*. Cambridge University Press, Cambridge, UK, 2003.

55. Vachharajani, M., Vachharajani, N., Penry, D.A., Blome, J.A. and August, D.I., Microarchitectural exploration with Liberty. in *35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)*, (2002), 271-282.

56. Wang, A., Killian, E., Maydan, D. and Rowen, C., Hardware/software instruction set configurability for system-on-chip processors. in *Design Automation Conference (DAC)*, (2001), 184-188.

57. Wunderlich, R. and Hoe, J.C., In-System FPGA Prototyping of an Itanium Microarchitecture. in *Internation Conference on Computer Design (ICCD)*, (October, 2004).