UNIVERSITY OF CALIFORNIA
Santa Barbara

Synthesis of Synchronous Pipelined Circuits from High-Level Modular Specifications

by

Maria-Cristina Marinescu

A dissertation submitted in partial satisfaction of the
requirements for the degree of

Doctor of Philosophy

in

Computer Science

Committee in charge:

Professor Martin Rinard, Chair
Professor Oscar Ibarra, Chair
Professor Omer Egecioglu
Professor Anurag Acharya

December 2002

The dissertation of Maria-Cristina Marinescu is approved:

Professor Anurag Acharya

Professor Omer Egecioglu

Professor Oscar Ibarra, Chair

Professor Martin Rinard, Chair

April 2002

Curriculum Vitae

Maria-Cristina Marinescu

EDUCATION

Visiting Scholar, **Massachusetts Institute of Technology**
Laboratory for Computer Science, August 1997 - September 2002

PhD in Computer Science, **University of California, Santa Barbara**
Department of Computer Science, December 2002 (expected)

B.S. in Computer Science, **Politehnica University of Bucharest**
Department of Computer Science, June 1995

PROFFESIONAL APPOINTMENTS

Summer Internship, **Bell Labs, Lucent Technologies**, Murray Hill NJ
Nevin Heintze, June 2000 - September 2000

Research Assistant, **Massachusetts Institute of Technology**, LCS
Prof. Martin Rinard, August 1997 - April 2002

Research Assistant, **University of California at Santa Barbara**, Computer Science Department
Prof. Martin Rinard, June 1997 - September 1997

Teaching Assistant, **University of California at Santa Barbara**, Computer Science Department
September 1995 - June 1997
Courses in: Operating Systems, Networking, Theory of Computation, Data Structures and Algorithms, Parallel Scientific Computation

SELECTED PUBLICATIONS

Maria-Cristina Marinescu and Martin Rinard, "A Synthesis Algorithm for Modular Design of Pipelined Circuits", Proceedings of the X IFIP International Conference on VLSI, Lisbon, Portugal, Dec. 1999.

Maria-Cristina Marinescu and Martin Rinard, "High-level Specification and Efficient Implementation of Pipelined Circuits", Proceedings of ASP-DAC Asia and South Pacific Design Automation Conference 2001, Yokohama, Japan, Jan.-Feb. 2001.

Maria-Cristina Marinescu and Martin Rinard, "High-level Synthesis of Pipelined Circuits from Modular Queue-Based Specifications", IEICE Transactions, Special Issue on VLSI Design and CAD Algorithms, 2001.

Maria-Cristina Marinescu and Martin Rinard, "High-level Automatic Pipelining for Sequential Circuits", Proceedings of the 14th International Symposium on System Synthesis, ISSS 2001, Montreal, Canada, Sept.-Oct. 2001.

FIELDS OF STUDY

Major Field: Computer Science

Studies in: High-level Hardware Synthesis
Program Analysis
Compilation Techniques

Advisor: Martin C. Rinard

To Costin, who silently taught me everything

I am, and everything I hope to become.

Abstract

Synthesis of Synchronous Pipelined Circuits from High-Level Modular Specifications

by

Maria-Cristina Marinescu

Digital circuits have become increasingly more complex at a steadily faster rate over time. As a result, the complexity of the challenge posed to designers to produce efficient, reliable circuits of larger size in less time is growing. This thesis presents a specification language and a synthesis system whose goal is to reduce the amount of time and effort required to design digital circuits and widen the range of people that can build circuits without having to become expert hardware designers.

Our approach relies on novel compiler technology to synthesize highly concurrent, synchronous implementations of circuits, starting from modular, sequential, asynchronous specifications written in our specification language. The modularity and asynchronicity properties of our language make it possible for the designer to think about each module in isolation, which substantially reduces the design time and effort. These properties also enable locality of the specifications, which facilitates developing, debugging, formally verifying and reusing either partial or complete specifications.

Our system is designed with two goals in mind: ease of design and high circuit performance. After experimenting with specifying circuits in our specification language, we can say that developing descriptions using this language approach rather than using Verilog is significantly faster, and the size of the specifications considerably smaller. We have implemented a compiler that takes a description in this specification language and automatically generates a synchronous, concurrent implementation of it in synthesizable Verilog. Rather than having the designer manually coordinate the simultaneous execu-

tion of various parts of the circuit, we make it the compiler's job to discover the concurrency in the specifications and expose it in the final implementations. Our algorithms preserve the correctness of the original specifications. After comparing clock cycle and circuit area numbers for our benchmarks with the numbers obtained from equivalent manually-coded Verilog benchmarks, we conclude that our automatically generated circuits provide performance that is virtually identical to the performance of manually written versions. As an important application, we developed an algorithm for efficient, automatic pipelining of unpipelined or insufficiently pipelined circuit descriptions, which implements techniques like stalling and forwarding.

Contents

List of Figures

Acknowledgements

Anybody that completed a PhD must be familiar with the question:

"How do you feel, now that you're all done?"

When I was a kid my parents would put me in the family car and drive, sometimes for days, seeing places—sometimes we'd stop, but never for too long; we kept driving. I'm not the kid that said "Are we there yet?" It was always the journey that was the destination. Destinations were good as long as they grew new beginnings. Finishing my PhD journey marks the beginning of a next journey to come. What I want to remember are the paths, their forks and joins, and the people I came upon on the roads, and at intersections. People who helped me, or whose mere presence made it worthwhile to keep walking the road towards PhD completion.

Martin C. Rinard: For believing in me, for all the technical and professional advice, for always making time, for support and open-mindness in all respects, I feel most fortunate to have had you as my advisor.

Costin C. Iancu: For being.

Maria & Valeriu Marinescu: For all the countless sacrifices, I hope I can justify them.

Darko Marinov: For true friendship, many technical discussions, and for teaching me formal program verification, for putting up with being my office-mate and occasionally driving you crazy, I don't wish I had a brother like you, since you are already my friend.

`freepizza@lesser-magoo.lcs.mit.edu:`

Chandra Boyapati, Karen Zee: For spending time to show to the idealist in me how being more pragmatic can sometimes take you far.

Viktor Kuncak: For lots of helpful and interesting technical discussions, and for being a friend.

Monica & Radu Rugină: For making my life better and more normal, for Monica's soups, Radu's soccer, and for getting drunk with me (and other friends) until 7am on the day of your wedding.

Scott Ananian, Brian Demsky, Patrick Lam, Alexandru Sălcianu: For creating a rich technical environment, and for being a happy research group, with more hobbies than that of spending the nights at the lab.

Srinivas Devadas: For very useful technical discussions and genuine support.

Nevin Heintze: For making even NJ (and the NJ Summer) worth putting up with—the interesting work, the good atmosphere, the understanding for getting my own paper out while interning, and bocce.

Saman Amarasinghe: For professional support and encouragement.

Anurag Acharya, Ömer Egecioglu, Oscar Ibarra: For taking time to understand and evaluate my research.

Arvind , Krste Asanović , Urs Hölzle , Stefanos Kaxiras , Brian Kernighan , Balaji Prabhakar: For fun, interesting conversations.

Cornelia Colyer: For making Fedex Overnight remember my name out of all the others.

Shireen Agah: For educating and encouraging me into painting.

Maryjane Archenbronn: For making it easy for me to be in Boston while still a UCSB student.

Rachel Allen: For being such a friend, despite my poor maintenance skills.

Alejandro Caro, George Hadjiyiannis, Sarfraz Khurshid, Sajit Rao: For the friendship.

Albert Alexandrov, Pedro Diniz, Max Ibel, Denis Khotimsky, Roxana Stanoi, Constantin Vasiliu: For helping me out to have a smooth transition to the US, and for being big part of the reason I now call Santa Barbara home (although only Albert still lives there).

Corneliu Barbu: For driving from work in LA to catch my defense in Santa Barbara (but not only this).

Mihai Ionescu: For getting me started on the road to the US and to graduate school.

Mathias Kölsch: For making it possible to write this section and send you myThesis.ps, instead of flying the red-eye tonight.

Dawson R. Engler: For the inspiration.

Chapter 1

Introduction

The goal of the research that I will present in this thesis is to reduce the amount of time and effort required to design digital circuits and widen the range of people that can build circuits without having to become expert hardware designers. As systems grow larger and more complex and require better performance and enhanced reliability, the process of using existing design tools to build circuits becomes more complicated. Increased automation is desirable because it simplifies this process and holds out the promise of reducing the time and cost of developing circuits. We have developed a new approach to designing systems that we believe makes the design of digital circuits easier and more efficient. This approach relies heavily on novel compiler technology capable of delivering highly concurrent, synchronous implementations of circuits described using our method. We have designed a specification language that reflects our approach and implemented a compiler able to automatically synthesize efficient, fully pipelined circuits from a description given in this language.

The rest of this introductory chapter briefly presents the basic ideas behind our specification language, identifies some of its key features, compares our approach with other existing approaches, and outlines the contributions of this thesis.

1.1 Our Basic Approach

We specify a circuit in our language as state and computation; we specify the state through state declarations. There exists a state declaration for each piece of state used by the circuit. The state is implemented in hardware as registers and memories and is of either a primitive type, a tagged union type, an array type, or a queue type. The queue type represents conceptually unbounded first-in first-out (FIFO) queues.

A developer using our language defines the computation as a set of update rules. Each update rule contains two parts: an enabling condition and a set of updates to the state. The enabling condition tests the current state of the system to determine whether the rule is enabled; if the truth value of the enabling condition is false, the rule is not enabled for execution, otherwise it is. Each rule update contains a set of actions, where each action contains an expression that computes the new value of a state variable, and an assignment of this value to the state variable. If a rule is enabled, it can execute. When it executes, its set of actions updates the current state of the system.

In our system, rules execute atomically, sequentially, and asynchronously with respect to each other. Each rule is atomic in the sense that all of the updates performed by the rule finish before any other rule can execute. Conceptually, the execution of a system repeatedly chooses an enabled rule and completely executes it before choosing another rule. Because of this reason, we say that our system has sequential semantics.

A module definition consists of a set of update rules. Modules interact with other modules by reading and writing shared state; queues are a particularly relevant kind of state. Typically, modules read inputs from some input queues and a generally small set of other state (registers, memories), perform some computation, then write values back into their output queues and a subset of the state. The queues decouple the modules and therefore the execution of the rules in the modules. The execution of a rule depends only on the values that it reads from its input queues and other state variables and is otherwise

decoupled from the execution of the other rules. We therefore say that the modules (and rules) execute asynchronously with regard to each other.

Our compiler takes a specification of a circuit written in this language and automatically composes the module definitions to generate a global schedule for all of the operations in the rules. This schedule enables the synchronous and concurrent execution of multiple rules per clock cycle and produces a fully pipelined circuit implementation in synthesizable Verilog.

1.2 Our System

After designing, implementing and using our system, we consider that the important features of our specification language and compiler are:

- **The specification language provides atomicity via sequential semantics.** The execution of each rule is sequential at the specification level. This ensures that there is only one rule that executes at any time, and that no other rule declared in a different part of the specification can overlap or interleave its updates with those of the currently executing rule. We can view this property as a way to provide modularity to groups of actions. Atomicity is important because it allows the developer focus on one rule at a time when reasoning about the behavior of the system, without the need to think about the concurrent execution of the rules. On the other hand, an efficient implementation needs to be highly concurrent. The compiler therefore automatically discovers and exposes the parallelism in the final implementation.

- **Conceptually unbounded FIFO queues decouple modules in the specification.** Using unbounded queues to connect the modules in a specification enables the designer reason about and develop each module in isolation, without the need to think about global timing issues in the circuit. In that sense, queues decouple the temporal aspects of the design. The locality achieved by modular design facilitates debugging

and verification. Moreover, modularity supports the reusability of whole or parts of existing designs. Using update rules to specify the behavior of a circuit provides an intuitive, concise way of achieving a decoupled modular framework.

- **Starting from a modular and sequential specification, the compiler generates a final circuit implemented as synchronous logic.** A novelty of this approach is that while the resulting circuit is synchronous, the input specification has asynchronous semantics. It is the compiler's job to bridge this gap efficiently. Each of the rules that specify a system preserves the key invariants of the system and makes some kind of forward progress. The synthesis algorithm extracts the concurrency from the specification, schedules all of the actions in the rules, and specializes the circuit to execute that schedule.

1.2.1 Application Domain

We have designed our system to support the design and synthesis of standard synchronous digital circuits. We believe that our approach is especially well-suited to describing systems, such as pipelined circuits, that are naturally specified as a composition of interacting sub-systems. Other types of circuits that are a particularly good fit for our approach are DSPs and circuits for embedded systems. We have experimented with our system and found that our expectations are confirmed for circuits in the aforementioned classes. We expect our system to work well for synthesizing circuits in other classes, as well.

Sometimes, designers want to specify explicit timing or latency constraints between different operations in the circuit. In this case, our system needs to interface with other components that implement a timing constraint mechanism. The absence of a built-in timing constraint mechanism does not generally affect the strength of our compiler to generate efficient implementations for DSPs, embedded systems and pipelined processors. For the cases in which the detail

level of our language is not the best fit, our framework is still a very useful tool in experimenting with different prototypes of these circuits.

1.2.2 Advantages

Within its application domain, our system is designed to provide a simple, high-level way to specify a circuit without sacrificing the efficiency of the resulting implementation. The decoupled semantics of the specification allows the designer to develop one module at a time and compose the modules together without reasoning about their concurrent execution. The locality that asynchronicity enables makes specifications easier to write, debug and formally verify; it also enables module reuse.

The sequential semantics of our system provides atomicity of groups of actions and makes the specification process less prone to mistakes and easier to understand. The compiler later discovers the concurrency in the specification and exposes it in the final implementation.

1.2.3 Evaluation

Our system is designed with two goals in mind: ease of design and high circuit performance. To evaluate how well these challenges are met, we first developed a set of benchmarks written in our specification language, then passed them through our compiler to generate circuits described in synthesizable Verilog. We used the applications experience that we gathered to quantify how well we accomplished our initial goals:

- **Ease of design:** We address this requirement by giving qualitative measures of time to write the circuit specification, specification size and specification-to-Verilog synthesis time. After experimenting with our system, we concluded that developing specifications using our language rather than using Verilog is significantly faster and the size of the specifications is about one or more orders of magnitude smaller. For

our smaller benchmarks, the running time of the compiler is negligible, but it becomes noticeable for designs with complicated control circuitry. Compilation time can become an issue if the circuit is being modified repeatedly or developed interactively.

- **Performance:** This measure gives quantitative evidence of how well the resulting implementations perform in terms of clock cycle and circuit area. After comparing our numbers with the numbers obtained from equivalent manually-coded benchmarks we conclude that our automatically synthesized circuits provide performance that is virtually identical to the performance of manually written versions.

Our experience with the system leads us to the conclusion that our system makes the experience of building circuits a lot easier than designing circuits manually as HDL specifications. Even for experienced designers, having to permanently reason about concurrency and the coordinated actions of a complex system is sometimes a painful experience and often an error-prone process. Our framework provides a high-level, asynchronous and sequential way of designing a system that makes circuit design an easier, more manageable task for a much larger class of users.

1.3 Other Approaches

This section briefly introduces various existing approaches to hardware synthesis and underlines the difference between these and our approach.

1.3.1 Synchronous vs. Asynchronous Communication Paradigm

Most existing systems provide a framework and a specification language that fit a synchronous approach. Processes in VHDL [7], Verilog [86] or Scenic [39] communicate via signals. Synchronous languages like Esterel [16],

Lustre [44], Signal [43] and Statecharts [45] display the classical FSM behavior; here communication is realized by instantaneous broadcasting. The disadvantage of using a language with synchronous semantics, even if the language supports modular designs, is that it forces the designer think about global timing issues when describing the system. This is true because the language cannot achieve the complete locality that allows one to think about each module in isolation from the rest of the system.

Other systems embrace an asynchronous communication paradigm. Estelle and SDL are languages designed for protocol specification and simulation and both use asynchronous FIFO queues to communicate between building blocks. Though asynchronicity enables modular development of the system's building blocks, concerns regarding how the details of the implementation phase would be handled, the speed at which designs could be created and the ability to include legacy code make most of the system specification languages not suitable for designing hardware.

1.3.2 Synchronous vs. Asynchronous Circuitry

If we look around, we are surrounded by electronic devices containing digital circuits. Almost all digital circuits we have a use for in everyday life are synchronous circuits. Why is there such a big gap between synchronous and asynchronous circuits?

The asynchronous design community argues that there are important advantages of asynchronous circuits; to mention a few of them: no clock skew worries, lower power consumption, average-case rather than worst-case performance, better technology migration potential. While this is a valid point which makes asynchronous implementations an interesting target, asynchronous design raises challenging problems that do not appear in synchronous design.

Completion signaling and detection requires extra time, which increases the theoretically average-case delay. Encouraging results have been obtained for very fine pipelines [67]. However, pipelining was done manually and choos-

ing the set of transformations was rather an ad-hoc process which is hard to be automatically reproduced. To obtain maximal throughput, data may need to be spaced through the pipeline; these transformations require the introduction of new buffer stages, which increases the circuit area. Asynchronous circuits also need non-multiplexed multi-ported memories, which get quickly more expensive with the number of ports, both in area as well as in propagation delay. Since there is no clock to synchronize with, designers usually multiplex ports by implementing some kind of a synchronization protocol for the memory accesses, which defeats the point of an asynchronous implementation.

The point can be made that part of the relative failure of asynchronous technology is due to the very fact that people have focused mainly on designing synchronous circuits. This tendency was initially the result of the synchronous abstraction being easier to fully understand and work with. Today, the main persisting appeal of the synchronous approach is exactly that it allows the designer to use the rich existing technology and interoperate with existing systems and designs. It is not clear long-term which of these two approaches is best and whether asynchronous circuits will become widely used in the future or not.

1.3.3 How Is Our Approach Different?

The main novelty of our approach is that it provides a specification language with asynchronous communication semantics and a compiler that is capable of generating synchronous circuitry from specifications written in this language. Connecting modules by unbounded FIFO queues is an elegant, implicit way to express concurrency and communication naturally without the need to use explicit synchronization or to introduce language extensions, as most of the software language-based approaches do. The modularity and asynchronicity properties of our language make it possible for the designer to think about each module in isolation; as a result the design time and effort are substantially reduced. The compiler transforms specifications written in such a

modular, decoupled fashion into highly-concurrent, fully synchronized circuit implementations. We can adapt our approach to automatically generate asynchronous implementations, assuming that such a circuit is desired as end-result of the synthesis process.

1.4 Contributions

We view the contributions of this thesis as consisting of:

- **The Approach:** It presents a new approach to high-level synthesis. This approach combines the best of both worlds: a modular, asynchronous, sequential specification language and an automatically generated synchronous, fully pipelined, concurrent implementation. It also presents a novel approach for automatically pipelining sequential circuits by repeatedly extracting a computation from the critical path and moving it into a new stage. This stage uses speculation to generate a queue of values that keep the pipeline full. This approach reduces the clock cycle and increases the throughput of a circuit.

- **The Algorithms:** It presents a relaxation algorithm for decreasing the clock cycle time and increasing the concurrency of the resulting circuit. It also describes a coordinated global scheduling algorithm for mapping the individual operations of the modules into clock cycles. This algorithm is the enabling technology for efficient pipelining, as it allows the data to move together across the circuit even when the pipeline buffers are full. The applications chapter introduces our pipelining algorithm and two extensions to the approach: stalling, which reduces the amount of area that would otherwise be required to respond to incorrect speculations; and forwarding, which increases throughput either by replacing values produced by incorrect speculations with correct values or by making new values available earlier to the stall logic.

- **The Evaluation:** It presents experimental results that demonstrate the effectiveness of the technique in practice.

1.5 Organization

The rest of the thesis is organized as follows. Chapter 2 presents a simple example that illustrates our approach. Chapter 3 describes our specification language, a language with asynchronous, atomic and sequential semantics. The chapter ends with a discussion on the advantages and limitations of using this approach. Chapter 4 presents the implementation of our system and describes the optimizations that the compiler applies to produce efficient circuits. Chapter 5 presents the formal framework and detailed correctness proofs for the main algorithms that our compiler implements. Chapter 6 describes our experience building a set of benchmarks and automatically generating circuit implementations in synthesizable Verilog. For evaluation purposes, the area and clock cycle of the resulting implementations are compared against Verilog applications developed by hand. Chapter 7 presents an important extension to our synthesis algorithm, namely an automatic pipelining algorithm for sequential circuits. Chapter 8 presents a summary of some of the most well-known system level specification languages and high-level synthesis systems. Chapter 9 discusses several future work extensions and Chapter 10 presents the conclusions.

Chapter 2

Example

In general, circuits contain state and computation. The state holds values across clock cycles, is distributed throughout the circuit, and is implemented as either hardware registers[1] or memory. The computation is implemented as combinational logic that transforms data during each clock cycle. Our language enables the designer to specify the computation as a set of *modules*. Each module performs local computation and interacts with other modules by reading and writing shared state.

In pipelined circuits, data flows from one pipeline stage to another through pipeline registers. Our language enables the designer to group the operations in each pipeline stage into a separate module and connect these modules using FIFO queues. Our synthesis system automatically translates the FIFO queues in the specification into pipeline registers in the circuit implementation.

We illustrate this approach by presenting a simple example: a linear pipelined datapath with associated control, which implements a very reduced instruction set: an `INC` instruction, which increments the value

[1]The basic hardware component that has the capability to store a value is the register. At a higher, architectural level, a register is an entry in a register file. To avoid confusion, for the rest of the thesis, we will use the term register for hardware register and the term architecture register for the register file entries.

in its single register argument, and a `JRZ` instruction, which tests the value in its register argument and, if the value is zero, jumps to the location in its location argument. We next present the specification for a three-stage pipelined implementation of this instruction set.

Our basic approach is to specify the circuit as consisting of its state and the modules describing the actions that manipulate the state. Each state variable has a data format. The designer specifies the data format for all state variables either using separate type declarations or within the state declarations themselves. To keep a minimal, clean interface between modules, we chose to build one module per pipeline stage. Therefore, we distribute the execution of each instruction over the pipeline stages and end up with three modules: an instruction fetch module IFM, a register operand fetch module ROFM and a compute and write back module CWBM.

2.1 State

```
1 type reg = int(3), val = int(8), loc = int(8);
2 type ins = <INC reg> | <JRZ reg loc>;
3 type irf = <INC reg val> | <JRZ val loc>;

4 var pc :  loc, im :  ins[N], rf :  val[8];
5 var iq = queue(ins), rq = queue(irf);
```

Figure 2.1: State Variables and Type Declarations for Example in Fig. 2.2

Lines `1` through `3` in Fig. 2.1 present the type declarations that define the format of the data in the circuit. Each type declaration consists of:

- The keyword `type`.
- A type name.
- The equal sign `=`.

– A type definition.

Following this pattern, line 1 declares three type names:

– The type name `reg`, which denotes a 3 bit integer value. Our specification uses this type to represent architecture register names.

– The type name `val`, which denotes an 8 bit integer data type and is used for the values in the register file.

– The type name `loc`, which denotes an 8 bit integer data type. This type represents the locations of the instructions in the instruction memory.

In our example circuit, `INC` instructions and `JRZ` instructions have different formats. To represent a data type with several different formats — in our case instructions — we introduce a tagged union type, similar to those found in ML [71] and Haskell [53]. Line 2 declares a tagged union type `ins` which represents instructions. An `ins` type instruction can have of one of two data formats:

– Increment instructions have an `INC` tag and a register name field of type `reg`.

– Jump if register value zero instructions have a `JRZ` tag and two fields: the register name of type `reg` and the jump location of type `loc`.

Line 3 declares the type `irf` for instructions whose register operands the processor has already fetched from the register file. This type declaration is also of tagged union type. The instructions of type `irf` are either:

– Increment instructions, which have an `INC` tag and two fields: a register name of type `reg` and a register value of type `val`.

– Jump if register value zero instructions, which have a `JRZ` tag and two fields: the register value of type `rval` and the jump location of type `loc`.

Our synthesizer represents data types in hardware as follows. For simple types like integer or char, the representation is straightforward. Each type declaration has an associated width of `n` bits; the type is represented as an `n`-bit register or wire. For tagged union types, our synthesis system computes the number of bits for the tag field and associates a unique integer with each tag. It also computes the resulting size in bits of all the fields in the tagged union type, including the tag itself. This number is the size in bits of registers that contain values of the tagged union type and of the wires that transmit values of this type.

Line `4` and `5` present the state declarations. Each state declaration consists of the following:

– The keyword `var`.

– The state variable name.

– A semicolon `:`.

– The type of the state variable.

Line `4` declares the following state variables:

– A program counter `pc` of type `loc` declared on line `1`.

– An instruction memory `im` of type array of `N` instructions of type `ins`.

– A register file `rf` of type array of `8` values of type `val`.

The declarations on line `5` use a predefined `queue` data type. A `queue` is a conceptually unbounded first-in first-out (FIFO) queue that carries values between modules. The specification language includes a number of

primitive operations on queues. We implement queues in the hardware as a number of registers equal to the length of the queue. Line `5` in Fig. 2.1 declares:

- A queue `iq` of instructions of type `ins`, for fetched instructions.
- A queue `rq` of instructions of type `irf`, for instructions whose register operands have been fetched.

2.2 Modules

Our circuit executes a sequence of instructions. To execute each instruction, the circuit performs the following steps:

- It reads the instruction from the instruction memory.
- If the instruction is an `INC` instruction, it reads a value from the register file, increments it and stores it back into the register file.
- If the instruction is a `JRZ` instruction with the value in its register argument zero, it jumps to the location argument and continues execution from there.
- If the instruction is a `JRZ` instruction with the value in its register argument different from zero, it does nothing.

Our specification distributes the execution of each instruction over three pipeline stages. To keep the pipeline full, the circuit uses a speculative approach that starts the execution of the next instruction before it knows whether the instruction should execute or not. If the circuit starts executing an instruction but later determines that the instruction should not execute, it restores the state of the circuit to reflect the values before the instruction started executing, then restarts the execution from the correct program counter. In this case, speculative execution

does not increase the throughput of the circuit. In fact, it even requires additional space to save the state that needs to be recovered if a speculation goes wrong. But if the speculation was correct, the circuit performs useful computation for each clock cycle that otherwise would have been spent stalling. Pipelining combined with speculation can increase the parallelism in the system and therefore boost the throughput of the circuit. We consider that the advantages of a speculative approach makes it worth using in our specification.

For our circuit, we have one module for each pipeline stage:

- IFM, the instruction fetch module.
- ROFM, the register operand fetch module.
- CWBM, the compute and write back module.

The queue `iq` connects IFM and ROFM; the queue `rq` connects ROFM and CWBM. Fig. 2.2 shows the three functional modules in our example and the queues that interconnect them. Modules interact with other modules by reading and writing shared state. Each module definition is composed of one or more rules. Rules read values from input queues and other state variables, perform local computations, and write results to output queues and other state variables. A rule is enabled if its enabling condition evaluates to `true` in the current state. Conceptually, the execution of the system repeatedly chooses an enabled rule and executes it. We illustrate the conceptual model of execution in our system by discussing the operation of the rules in our circuit.

2.2.1 The Instruction Fetch Module

We need only one simple rule to describe the functionality of the IFM module:

```
true → iq = append(iq, im[pc]), pc = pc + 1;
```

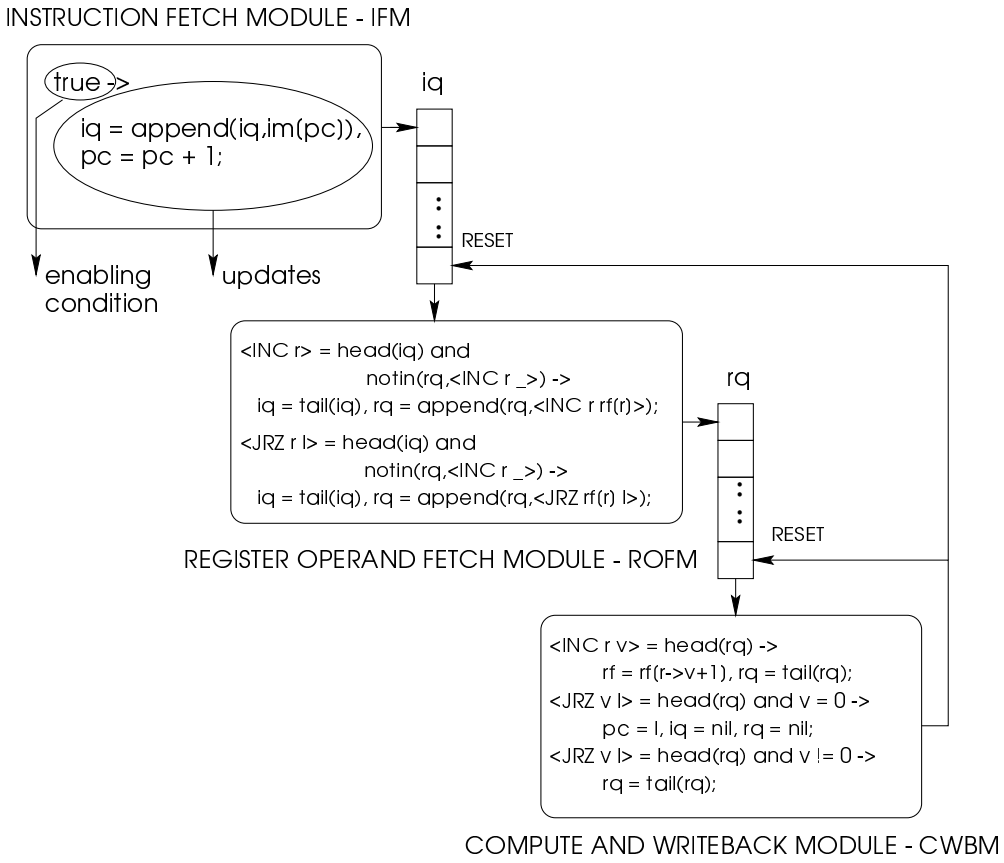

Figure 2.2: Specification Example

In general, each rule consists of two parts separated by an arrow. The first part of the rule — on the left-hand side of the arrow — specifies all of the preconditions that need to evaluate to true in the current state for the rule to execute. The precondition for our rule is always `true`, which means that the rule is always enabled.

The second part of the rule — on the right-hand side of the arrow — describes all the actions that the rule will execute if the preconditions are all fulfilled. All the actions in the right-hand side of the arrow execute atomically, together. Because our rule is always enabled, it will be always ready to execute. When it executes, it fetches an instruction from the instruction memory and inserts it into the instruction queue `iq`. It also increments the program counter `pc` to set up the next fetch.

Recall that the pipeline has to preserve the appearance of sequential execution. When a `JRZ` instruction executes, it tests the value in its register argument. If this value is zero, the branch is taken and the execution continues from the location argument of the `JRZ` instruction; otherwise it continues from the next value of the program counter (`pc + 1`). Therefore, having the IFM fetch the next instruction while a `JRZ` instruction is still in the pipeline may violate the sequential execution semantics we want to preserve. Because the datapath fetches instructions speculatively, we need another rule that, when a speculation is incorrect, restores the correct state of the circuit and restarts the execution from the correct program counter. We will see how we can specify such a rule when describing the CWBM.

2.2.2 The Register Operand Fetch Module

The two rules in the module ROFM remove instructions from `iq`, fetch the register operands, and insert them into `rq`. The first rule processes `INC` instructions, and the second one processes `JRZ` instructions. Both

rules use a form of pattern matching similar to that found in ML and Haskell.

```
<INC r> = head(iq) and notin(rq, <INC r _>) →
        iq = tail(iq), rq = append(rq, <INC r rf[r]>);

<JRZ r l> = head(iq) and notin(rq, <INC r _>) →
        iq = tail(iq), rq = append(rq, <JRZ rf[r]
l>);
```

The enabling condition of the first rule is `<INC r> = head(iq) and notin(rq, <INC r _>)`.

- The first clause of the precondition, `<INC r> = head(iq)`, requires that the instruction at the head of the instruction queue `iq` be an increment instruction with register name `r`. If this is true, the clause matches and *binds* the variable `r` to the register name argument of the `INC` instruction, to be used later in the rule when referring to this operand.
- The second clause, `notin(rq, <INC r _>)`, uses the binding to test that the queue `rq` does not contain an increment instruction whose first argument is `r`. The second argument is irrelevant; we denote this using the _ sign. This clause implements a test for a read after write hazard (RAW); if there is a pending instruction waiting to execute that will write the register `r`, the machine delays the operand fetch so that it fetches the value after the write (this translates into *stalling*). If there is a pending instruction that will write `r`, the instruction is in `rq`. The clause `notin(rq, <INC r _>)` checks to make sure that there is no such instruction in `rq`. We only need to check `INC` instructions because `JRZ` instructions never write a location in the register file.

The rule as a whole is enabled and can execute only if the next instruction in `iq` is an `INC` and there is no RAW hazard. If enabled, the rule atomically executes the block in the right-hand-side of the arrow. The second rule also reads the register file and therefore performs similar actions for `JRZ` instructions.

2.2.3 The Compute and Write Back Module

The CWBM consists of three rules: one handles `INC` instructions and two handle `JRZ` instructions.

```
<INC r v> = head(rq) →
        rf = rf[r->v+1], rq = tail(rq);

<JRZ v l> = head(rq) and v = 0 →
        pc = l, iq = nil, rq = nil;

<JRZ v l> = head(rq) and v != 0 →
        rq = tail(rq);
```

- The first rule reads an `INC` instruction from `rq` and updates the index location value of the register file to the incremented value of that location. It also removes the instruction from `rq`.
- The second rule treats the case when the instruction at the head of `rq` is a jump instruction and the value in its register argument is zero. In this case, our system performs three actions. It first restores the state to reflect its values as if the datapath had not subsequently fetched any instruction following the `JRZ` instruction. It then flushes all the rules following the `JRZ` rule, that it already fetched from the instruction memory. Finally it restarts the execution of the system from the new target location. In our case, the only state variables that were modified because of subsequent

fetches from `im` are `pc` and `iq`. But `pc` gets updated to `1`, so no save/restore schema is necessary for `pc`. Queues `iq` and `rq` are both flushed, which makes saving and restoring `iq` unnecessary as well.

– The third rule of the CWBM handles untaken `JRZ` instructions. The rule simply removes the first instruction in `rq`.

The execution of our system is a sequence of rule executions. At each clock cycle, each rule gets a chance to execute. The system repeatedly chooses one of the enabled rules and atomically executes it.

2.3 Discussion

The key concepts of our specification are state and modules. A particularly useful kind of state is of `queue` type. Queues are conceptually unbounded. Grouping operations in modules connected by conceptually unbounded queues enables the designer think about each module in isolation, without the need to synchronize the computations in different modules with each other.

Modules consist of one or more rules. Each rule has a precondition part and an action part. A rule is enabled when its precondition evaluates to true in the current state. When the rule is enabled, the updates in the action part all take place atomically. Atomicity ensures that the executions of any two rules do not interleave.

The high-level behavioral specification style provides a simple, intuitive, and concise way of describing the functionality of the system. Our synthesis system takes such a specification and automatically generates a synthesizable implementation in Verilog.

Chapter 3

The Specification Language

The idea behind our specification language is to provide a simple, high-level way to specify a system without sacrificing the efficiency of the resulting implementation. We believe that the language which we describe in this chapter meets this challenge. Our language satisfies both expressiveness and efficiency requirements as follows. From the designer's standpoint, the language provides a simple way to reason about and describe the behavior of the system. From the implementation's standpoint, the language supports the application of automatic optimization and synthesis techniques. An efficient implementation is often synchronous and the result of globally scheduling all of the operations in the system. Our specification language makes applying these kind of transformations natural and effective.

This chapter describes the specification language that meets our requirements. It describes the basic concepts of the language and the type of operations the language can express. It then analyzes the advantages and limitations of our design choice.

3.1 Basic Concepts

As illustrated in our example, our specification language is based on two basic concepts: state and modules. The designer first specifies all the type declarations that define the format of the data manipulated by the circuit. Next, he declares all the state variables that constitute the state of the circuit. Finally, he specifies all of the modules. Modules are relatively isolated entities which model actions that manipulate a typically small related set of state variables. Modules interact with other modules by reading and writing shared state.

The next few sections introduce the two fundamental concepts of our language.

3.1.1 State

The state of a system consists of all the state variables used to specify the circuit. Each state variable has some data format. The designer specifies each format of data used by the circuit in either the type declaration or the state declaration section. He first specifies all the type declarations, then declares all the state variables used in the circuit. The state is implemented in hardware as registers and memories and holds the values of all the state variables across clock cycles.

Type Declarations

In general, each type declaration consists of:

- The keyword `type`.
- A type name.
- The equal sign `=`.
- A type definition.

The type declarations below list all the data types supported by our language.

type *size* = **string** || **int;**
type *predefinedSimpleType* = **int** || **char;**
type *widthPredefinedSimpleType* = *predefinedSimpleType(size)*;
type *id* = **string** || **string(***size***);**
type *simpleType* = *widthPredefinedSimpleType* || *id*;
type *tuple* = <*id field*$_1$:*simpleType* ... *field*$_k$:*simpleType*> ;
type *tupleTaggedUnion* = *tupleTaggedUnion* | *tuple* || *tuple*;
type *basicType* = *simpleType* || *tupleTaggedUnion*;
type *arrayType* = *basicType[size]*;
type *queueType* = **queue***(basicType)*;

Our synthesizer represents each data type in hardware as either registers or memories, as follows:

- The data types *id* and *widthPredefinedSimpleType* can have an associated width in bits, say `n`. We represent these data formats as an `n`-bit register that holds values of the corresponding type(**int**, **char** or **string**) and as a wire that transmits such values.
- *tupleTaggedUnion* is a tagged union type, similar to those found in ML [71] and Haskell [53]. For tagged union types, our synthesis system computes the number of bits for the tag field of type *id* and associates a unique integer with each tag. It also computes the resulting size in bits of all the fields in the tagged union type, including the tag itself. This number is the size in bits of the registers that contain values of the tagged union type and of the wires that transmit values of this type.
- *arrayType* is the type that represents state variables implemented as synchronous memory arrays. For each *arrayType* variable in the specification, our system generates a synchronous memory array with *size* elements of type *basicType*. The memories can have more

than one write port if otherwise independent operations in the circuit can write different locations of the array in the same clock cycle. If the primitive array used in the implementation has fewer write ports than the number of write operations to that array at the same clock cycle, the synthesis algorithm must only allow a subset of these operations to take place. Our current implementation is targeted to maximize concurrency in the resulting circuit, and therefore it allows writes to different locations in an array to happen simultaneously.

– A *queueType* is a predefined type that represents FIFO queues of conceptually unbounded length. FIFO queues provide buffered, first-in, first-out typed connections between modules. Queues isolate modules, therefore allowing them to execute asynchronously with each other. In hardware, each queue must have a finite representation. The designer declares a physical queue length `l` for each queue type declaration. The declaration also contains the type of the elements composing the queue. The synthesizer determines the size `n` in bits of the queue element type and represents each queue type as `l` `n`-bit hardware registers.

State Declarations

The designer must specify a state declaration for each state variable used in the specification. In hardware, the state is represented as registers and memories and it holds the values of all the state variables across clock cycles.

In general, each state declaration consists of:

– The keyword `var`.

– The state variable name.

– A semicolon `:`.

– The type of the state variable.

The type of the state variable can be either a predefined type from the type declaration section or any other type definition. Below we give a few examples of type and state declarations.

To declare a state variable `age` of type `int(3)`, we specify first a type definition, then the state declaration, as shown below:

```
type years = int(3);
var age:  years;
```

Assume we want to declare a state variable `games` that keeps the yearly records of all the games played by a soccer player. Each element of the record has one of the following formats:

- `(WON date:int(2) numberOfGoals:int(2) title:id)`
- `(LOST date:int(2) numberOfGoals:int(2))`
- `(EQUAL date:int(2) numberOfGoals:int(2))`
- `(NOTPLAYED date:int(2))`

Each player plays at most 36 games a year. To declare such a state variable, we need the following type declarations:

```
type name = string(10);
type record = <NOTPLAYED date:int(2)> ||
       <LOST date:int(2) numberOfGoals:int(2)> ||
       <EQUAL date:int(2) ynumberOfGoals:int(2)> ||
       <WON date:int(2) numberOfGoals:int(2) title:name>;
```

We can now declare the state variable `games` in one of the three following ways:

```
var games :  record[years][36];
```

or

```
type oneYear = record[36];
var games :  oneYear[years];
```

or

```
type GAMES = record[years][36];
var games :  GAMES;
```

To declare a state variable `program` that contains soccer games played on a specific date, we will specify the following type and state declarations:

```
var program : queue(record);
```

or

```
type PROGRAM = queue(record);
var program :  PROGRAM;
```

3.1.2 Modules

Modules are relatively isolated entities which model actions that manipulate the state. Each module reads shared state, performs local computation, then writes results back into shared state. The designer specifies each module as a set of *update rules*. Each update rule models a piece of computation performed by the system. Our synthesis algorithm implements each rule as combinational logic that modifies the state during each clock cycle. An update rule has two parts: an enabling condition and a set of updates to the state. In the example below, the enabling condition consists of everything before the arrow, while the updates come after the arrow.

```
true -> iq = append(iq, im[pc]), pc = pc + 1;.
```

Modules interact with other modules by reading and writing shared state. Most of this interaction takes place when inserting and removing elements into and from queues. A state variable of type **queue** is a conceptually unbounded FIFO queue that transfers values between modules. Rules read values from input queues and/or other state variables

(registers, memories), perform local computations and write results to output queues and/or other state variables. In the example above, the rule has no input queue, reads `pc` and `im[pc]` and updates output queue `iq` and state variable `pc`. The unboundedness of the queues allows them to decouple the modules in the specification. This property enables the designer to think of modules as executing asynchronously and independently from each other. This is a key insight that makes it possible to develop each module in isolation.

We say that a rule is enabled and can execute if its enabling condition evaluates to `true` in the current state. In this case, its updates are atomically applied to the current state to obtain a new state. For our example, the enabling condition is always true, and hence the rule will always be enabled and ready to execute. When it executes, it appends the next instruction in the instruction memory `im` to the output queue `iq` and increments the program counter `pc`.

State Transformations

This section describes the transformations that the modules can perform on the state. We group these transformations around the three most general types of data supported by our specification language:

– **The** *simpleType* **Data Type:** In hardware, we represent a state variable of type *simpleType* as a register. The kind of operations a module can perform on such state variables are the following:

Arithmetic Operations:

- `'+'` : Add the values of the two registers representing the state variables.

- `'-'` : Subtract the values of the two registers representing the state variables.

- `'*'` : Multiply the values of the two registers representing the state variables.

- `'/'` : Divide the values of the two registers representing the state variables.

- `'%'` : Compute the value in the register modulo the given integer.

- `arsh`, `alsh`: Variable arithmetic shift (right, left) of the value in the first register by the value of the second register.

- `arrot`, `alrot`: Variable arithmetic rotate (right, left) of the value in the first register by the value of the second register.

Logic Operations:

- `'&&'` : Logic AND between two boolean expressions.

- `'||'` : Logic OR between two boolean expressions.

- `' ^ '` : Logic XOR between two boolean expressions.

- `' ~ '` : Negation of a boolean expression.

- `rsh`, `lsh`: Variable logic shift (right, left) of the value in the first register by the value of the second register.

- `rrot`, `lrot`: Variable logic rotate (right, left) of the value in the first register by the value of the second register.

Boolean Tests:

- `'=='` : Returns true if the two registers representing the state variables hold equal values. Otherwise returns false.

- `'!='` : Returns true if the two registers representing the state variables hold different values. Otherwise returns false.

- `'>'` : Returns true if the value of the register in the left-hand side is greater than the value of the register in the right-hand side. Otherwise returns false.

- `'>='` : Returns true if the value of the register in the left-hand side is greater or equal than the value of the register in the right-hand side. Otherwise returns false.

- `'<'` : Returns true if the value of the register in the left-hand side is less than the value of the register in the right-hand side. Otherwise returns false.

- `'<='` : Returns true if the value of the register in the left-hand side is less or equal than the value of the register in the right-hand side. Otherwise returns false.

Bit Composition:

{`O1, O2, ..., On`}, where `O1, O2, ..., On` are operands of type `simpleType` or created by the selection operation below. The composition operation returns the data obtained by applying concatenation on all operands `O1, O2, ..., On`, in this order.

Bit Selection:

`O[n], O[n1:n2]`, where `n, n1, n2` are integer numbers and `O` is an operand of type `id` or created by the composition operation above. The selection operation returns the content of operand `O` at bit position `n` or between bit positions `n1` and `n2`.

Test and Jump:

- `if`: Perform a boolean test on the condition and return the result of the operation on the "then"/"else" branch if the condition evaluates to true/false.

- `switch`: Test the value of the case variable and return the result of the operation on the branch whose test returned true.

- `'?'` : Shortcut for `if`.

– **The** *arrayType* **Data Type:** In hardware, we represent each state variable of type *arrayType* as a synchronous memory array. The set of operations a module can perform on a state variable of this type is:

- Load value from memory index: `memory[index]`.

- Store value into memory at index: `memory = memory[index -> value]`.

– **The *queueType* Data Type:** Queues provide buffered, first-in, first-out connections between modules. There are several operations that modules can perform on a queue `q`:

`- head(q)`: Retrieves the first element in the queue `q`, but does not remove it from `q`.

`- tail(q)`: Returns the rest of the queue `q` after the first element and updates `q` to reflect this new value.

`- append(q,e)`: Returns the queue `q` after inserting the element `e` at the end of `q`.

`- replace(e1,e2,q)`: Returns the queue `q` after replacing all entries matching `e1` by `e2`. We use the `replace` operation to specify a bypass (see Chapter 7 for details).

`- notin(q,e)`: Returns `true` if the element `e` is not in the queue `q`; otherwise returns `false`.

`- q = nil`: Resets the queue `q` to be empty.

3.1.3 Execution Model

In our abstract model of execution, all the rules of a system execute atomically, asynchronously and sequentially with regard to each other. Conceptually, the execution of the system repeatedly chooses an enabled rule and executes it. This is a standard model of asynchronous execution found, for example, in systems such as Unity [26] and term rewriting systems [8]. In our implementation, rules are considered for execution according to the textual ordering in the specification.

A rule is *atomic* in the sense that the executions of any two rules do not interleave at any time. Once a rule is enabled and ready to execute, all its updates take effect before any other rule starts executing.

Each rule's execution depends only on the values in its input queues and those of other state variables that it reads. The fact that the queues are

conceptually unbounded decouples the executions of rules from different modules: the actual computation performed by a rule is completely isolated from computations performed by other rules. We therefore say that the rules execute *asynchronously*.

We call our execution model *sequential* because conceptually, rules are considered for execution one at a time, regardless of whether data dependencies exist between them or not. At each step of the system, there is only one rule that either evaluates its enabling condition or executes, updating the state. We want to emphasize that this execution model is used primarily to facilitate reasoning about the abstract behavior and correctness of the system. It does not directly reflect the actions of the generated circuit, which executes computations from multiple rules in parallel.

3.2 Discussion

Any approach to designing a specification language involves a trade-off between the expressive power and the level of support it provides for its target class of applications. Our specification language is a high-level language with asynchronous and sequential semantics. It is best suited for specifying systems that are naturally described as a composition of interacting subsystems. In this section we discuss the advantages of using our approach for specifying systems in its target application domain and the limitations that our choices impose.

3.2.1 Advantages

The asynchronous semantics of the specification enables the designer to concentrate on developing one module at a time and to reason about

the correctness of the specification without reasoning about the concurrent execution of the composed modules. It also allows the designer to compose modules together into a complete system without the need to deal with complex global issues such as the coordinated assignment of operations to clock cycles. Adopting a specification style in which a system consists of loosely-coupled modules that exchange values through unbounded queues makes parts of the specifications or even whole specifications readily reusable. The locality that asynchronicity exposes makes specifications easy to modify, debug and formally verify. Our synthesis algorithm eliminates the potential inefficiency associated with a direct asynchronous implementation by automatically generating a coordinated global schedule for all operations in the system. This schedule is used to generate an efficient synchronous implementation in synthesizable Verilog.

The sequential semantics has the advantage of presenting the designer with a simpler way of thinking about the system's execution. A sequential model of computation makes specifications easier to write, less prone to mistakes and easier to understand. The compiler later discovers the concurrency in the specification and exposes it in the final implementation.

We also consider our specification language a good choice for our purpose because it describes the external behavior of a system rather than a particular implementation of it. This makes specifications more intuitive and concise and gives the compiler more opportunities to choose the best final implementation for the existing specification.

3.2.2 Limitations

Our framework does not provide a mechanism for specifying timing constraints. It is impossible to specify, for example, data rates for pipelines

or latency constraints. Also, the absence of an explicitly parallel abstract execution model may limit the freedom of the designer to specify systems in a highly optimized, concurrent fashion. Nevertheless, parallel execution can be expressed in terms of latency constraints. Adding such a constraint mechanism does not require any conceptual modifications to the existing framework.

High-level specification languages provide no control over the low-level implementation choices; our specification language is no exception. We can also see this limitation as an advantage if we think of it as giving more freedom to the compiler. It also makes the language accessible to a wider set of designers.

3.3 Summary

Our specification language is a high-level language with asynchronous and sequential semantics.

The basic approach is to specify a system as a set of independent modules connected by conceptually unbounded queues. This approach provides a simple way of specifying a system while not sacrificing the efficiency of the resulting implementation.

Chapter 4

The Implementation

Our synthesis algorithm adopts the following approach: it takes the asynchronous, sequential specification and converts it into a synchronous, parallel implementation. The idea is to automatically compose the module definitions to generate a global schedule for all of the operations in the rules. This schedule enables the synchronous and concurrent execution of multiple rules per clock cycle and produces a circuit that, when no hazards are present, reads and writes each queue in the same cycle. It implements each queue as a finite hardware buffer.

The alternative is to implement the system as a collection of asynchronous modules connected by queues. If we choose to implement queues as an asynchronous connection mechanism between independently operating modules, the system as a whole suffers from synchronization overhead as modules dynamically handshake to transfer data. To obtain better throughput, the data needs to be spaced through the pipeline such that the number of pipeline stages matches some optimal value. This transformation increases the circuit area. Multi-ported memories may also increase the area and propagation delay of the resulting circuit. Besides performance issues, we also want our designs to interoperate with existing designs and make use of current design methodologies.

Generally, an efficient implementation of a system produces a global schedule for all the operations in the system and is often synchronous. Also, most existing systems and methodologies are designed for synchronous circuit design. We therefore decided to also generate synchronous circuit implementations.

4.1 Overview

In this chapter we present our implementation decisions and describe the optimizations that the compiler applies to produce an efficient circuit. For reasons we present in Section 4.9, we decided to generate a synchronous implementation from the given specification. This implies that the compiler needs to generate, at the granularity of individual clock cycles, a detailed global schedule for all of the operations in the rules, including the insertions into and removals from the queues. Our queue finitization algorithm is the enabling technique for deriving a synchronous implementation from the asynchronous specification of a system. In the initial specification, queues have unbounded length. But the hardware implementation must have a finite, specific number of entries allocated for each queue. Our compiler must make sure that for any execution instance of the system, none of the queues will overflow. We have developed two algorithms for generating implementations that preserve this property: one for acyclic specifications, and a second one, more complex, for cyclic specifications. The algorithm for the cyclic case is a generalization of the one for the acyclic case. The difference is that, for acyclic specifications, the compiler considers rules in isolation, while for cyclic specifications it looks at groups of rules that must execute together to maximize the number of executing rules per clock cycle and avoid both deadlock and overflow of the queues in the system.

Our implementation executes multiple rules at each clock cycle as follows.

Each set that consists of rules with no data dependencies within the set, can execute all its components in parallel. If a set of rules has data dependencies, a straightforward implementation still allows some or all of the rules to execute sequentially in the same clock cycle and modify shared state. The requirement is that the enabling condition of each rule evaluates to true for the value of the state variables produced by the previously executing rule. This condition is overly conservative; to fully exploit the concurrency available in the specification, our implementation transforms the rules, when possible, so that their enabling conditions test the state of the system at the beginning of the clock cycle rather than the state created by the previously executing rule. This transformation makes it possible to evaluate all of the preconditions immediately and in parallel rather than sequentially. To preserve correctness, the updates to each state variable still happen sequentially. We call this technique *relaxation*. The goal of relaxation is to decrease the clock cycle of the generated circuit.

The compiler also implements a set of techniques geared towards optimizing the generated combinational logic. These optimizations include common subexpression elimination and mutual exclusion testing. The former avoids unnecessary replication of hardware, while the latter eliminates false paths in the implementation.

Our approach targets the class of circuits that are naturally described as a composition of interacting sub-systems and whose behavior does not require specifying either (1) timing constraints or (2) explicitly parallel operations in the circuit. The designer can obtain this additional functionality by interfacing our system with a component which implements a timing constraint mechanism. Examples of good candidate applications for efficient synthesis are DSPs, embedded systems and pipelined processors. The application domain is even larger if the goal is prototyping or exploring the potential solution space.

The remainder of this chapter is structured as follows. Section 4.2 introduces the basic approach of our synthesis algorithm and the six phases that it consists of. The following six sections discuss each phase of the algorithm in detail. Section 4.8 discusses the target applications. Section 4.9 gives the motivation for our decision to generate a synchronous rather than an asynchronous implementation from a given specification. We summarize in Section 4.10.

4.2 Basic Approach

The basic approach is to give each rule an opportunity to execute at each clock cycle. The challenge is to ensure that the final result at the end of the cycle correctly reflects the sequential, atomic execution of all of the rules that execute in that cycle. The algorithm meets this challenge by symbolically executing the rules in sequence, with each rule operating on the output of the previous[1] rule. The derived expression for each state variable represents its new value at the beginning of the next clock cycle.

The algorithm consists of six phases:

- **Associate Versions With Each State Variable:** As a first step, the compiler orders all the rules[2] in the specification and computes the version of each state variable that each rule accesses. Symbolically executing the rules in some predetermined order ensures that the values of all the state variables at the end of the cycle correctly reflect the sequential, atomic execution of all the rules that were enabled and executed in the current clock cycle. The first rule will read version 0 of the variables and compute version 1. The second rule will read

[1] Previous and next refer to the textual ordering of the rules in the original specification.

[2] Our implementation uses textual ordering of the rules. Conceptually, any order would work, but the relaxation technique can be applied with better results if rules which insert elements into queues precede rules that remove elements from queues. The synthesis algorithm could include an initial phase that can, in many cases, order the rules so as to match the flow of data in the pipeline.

version 1 and compute version 2 and so on. By feeding the output of the previous rule into the next rule, we establish an initial schedule for symbolic execution.

- **Relaxation:** The specification produced by the operation performed in the previous step may suffer from an excessively long clock cycle, as the execution of the rules modifying shared state is completely sequentialized. The goal of the relaxation is to shorten the critical path within each clock cycle. Whenever possible, the algorithm relaxes the calculation of the enabling condition for each rule so that it is evaluated in the initial state (at the beginning of the clock cycle) rather than in the state created by the previously executed rule. To maintain correctness, the updates still execute sequentially if they operate on the same state variable. This transformation ensures that each element of data traverses at most one module per clock cycle, producing an acceptable critical path for the circuit. By increasing the parallelism in this way, we shorten the clock cycle of the circuit, and, indirectly, increase its throughput. Relaxation does not insert or remove delays in or from the circuit.
- **Global Scheduling:** In the initial specification, queues have unbounded length. But the hardware implementation must have a finite, specific number of entries allocated for each queue. Given a designer-specified length for each queue, the synthesis algorithm must generate an implementation that does not exceed that length. In the actual hardware, a given length of 1 for each queue translates into the synthesis of a standard pipeline. The obvious way to implement the finitization algorithm is to disable a rule whose enabling condition is true if any of the queues that the rule inserts into is full. We adopt a less conservative solution; if no hazards are present, the circuit can perform single or multiple reads or writes from and into each queue in the same clock cycle, even if the queues are initially full. The condition is that enough rules will execute that remove elements from queues,

therefore making space for new elements to be inserted. Queues can get arbitrarily large during the clock cycle as long as they are within the maximum specified length at the end of the cycle. This transformation is geared towards increasing the throughput of the resulting circuit.

- **Symbolic Execution:** Next, the algorithm symbolically executes all of the rules in sequence. An expression is generated for each state variable; this expression reflects all of the possible updates of that variable for that clock cycle and represents the value of the variable in the next clock cycle. Since at synthesis time there is no way of knowing which subset of the rules will execute at each clock cycle, the expressions contain conditionals that test the preconditions of all the rules that can modify the corresponding state variables.
- **Optimizations:** The synthesis algorithm next applies a spectrum of optimizations whose goal is to avoid unnecessary replication of hardware and eliminate false paths in the implementation. These optimizations currently include common sub-expression elimination and mutual exclusion testing for the expressions derived at symbolic execution. If an expression contains a value that will never actually occur in practice because the conditions required to obtain that value are mutually exclusive, its computation is eliminated from the expression. The mutual exclusion testing is implemented using resolution [10] and a set of reduction and simplification rules.
- **Verilog Generation:** In the final phase we generate synthesizable Verilog for the optimized expressions in the previous step. Each array variable is implemented as a synchronous memory array. Queues are implemented as hardware registers. All other state variables are also implemented as hardware registers. The derived expression for each state variable evaluates to the new value that gets written back into the state at the beginning of the next clock cycle.

We next discuss each phase of the algorithm in turn. We will use the simple example in Chapter 2 to walk through each transformation, starting from the initial specification and resulting in the final implementation. For convenience, we repeat the complete original specification here, as we find it in Figure 2.2.

4.3 State Variable Versions

The execution of a synchronous circuit is a deterministic process. To match the execution model of our implementation to the execution of the circuit, our algorithm first numbers the rules in some order to determine the intermediate state in which each rule will be evaluated. Choosing a specific order in which rules are evaluated, and, if enabled, execute, confers deterministic semantics to our system. In our implementation, the sequencing order is the textual order of the rules in the initial specification. Conceptually, any order would work, but the relaxation technique can be applied with better results if rules which append elements into queues precede rules that remove elements from queues. The synthesis algorithm could include an initial phase that can, in many cases, order the rules so as to match the flow of data in the pipeline.

Figure 4.2 shows the specification in Figure 4.1 after associating versions with each state variable.

The transformation creates a single assignment for each newly derived state variable. This makes it possible for our compiler to symbolically execute all the rules automatically, in a deterministic fashion.

4.3.1 Trade-offs

Because rules have fixed priorities, the algorithm cannot guarantee that the maximum number of rules will execute at each clock cycle. On the

```
// Type Declarations
type reg = int(3), val = int(8), loc = int(8);
type ins = <INC reg> | <JRZ reg loc>;
type irf = <INC reg val> | <JRZ val loc>;

// State Declarations
var pc :  loc, im :  ins[N], rf :  val[8];
var iq = queue(ins), rq = queue(irf);

// Instruction Fetch Stage
1:  true -> iq = append(iq,im[pc]), pc = pc+1;

// Register Operand Fetch Stage
2:  <INC r> = head(iq) and notin(rq, <INC r _>) ->
 iq = tail(iq), rq = append(rq, <INC r rf[r]>);
3:  <JRZ r l> = head(iq) and notin(rq, <INC r _>) ->
 iq = tail(iq), rq = append(rq, <INC rf[r] l>);

// Compute and Writeback Stage
4:  <INC r v> = head(rq) ->
 rf = rf[r->v+1], rq = tail(rq);
5:  <JRZ v l> = head(rq) and v = 0 ->
 pc = l; iq = nil, rq = nil;
6:  <JRZ v l> = head(rq) and !(v = 0) ->
 rq = tail(rq);
```

Figure 4.1: Initial Specification Example

```
// Instruction Fetch Stage
1:  true -> iq1 = append(iq0,im[pc0]), pc1 = pc0+1;

// Register Operand Fetch Stage
2:  <INC r> = head(iq1) and notin(rq1, <INC r _>) ->
    iq2 = tail(iq1), rq2 = append(rq1, <INC r rf1[r]>);
3:  <JRZ r l> = head(iq2) and notin(rq2, <INC r _>) ->
    iq3 = tail(iq2), rq3 = append(rq2, <INC rf2[r] l>);

// Compute and Writeback Stage
4:  <INC r v> = head(rq3) ->
    rf4 = rf3[r->v+1], rq4 = tail(rq3);
5:  <JRZ v l> = head(rq4) and v = 0 ->
    pc5 = l; iq5 = nil, rq5 = nil;
6:  <JRZ v l> = head(rq5) and !(v = 0) ->
    rq6 = tail(rq5);
```

Figure 4.2: Specification After Rule Numbering

other hand, associating a linear ordering for the rules means a simpler abstract execution model with reproducible behavior. We will later see that this approach also reduces the complexity of our algorithms.

4.4 Relaxation

The second phase of our synthesis algorithm *relaxes* the enabling conditions of each rule, whenever possible. Applying this technique produces a circuit with a shorter clock cycle and increased throughput.

4.4.1 Basic Idea

The execution of a rule R can update state variables tested by a subsequent rule R'. If this is the case, then, without relaxation, R' has to wait for R to execute and update the state before testing its precondition. The idea behind the relaxation technique is the following: if we can prove that the execution of R will not disable the enabling condition of R', we can modify the precondition of R' to test the state before R executes. The update part of R' remains unchanged. We say that we relaxed the enabling condition of R'. This transformation destroys false data dependencies and exposes additional parallelism in the specification, reducing the length of the critical path of the circuit.

Based on this basic approach, the rest of the section presents our relaxation algorithm and outlines a proof of correctness. We then discuss the type of circuit optimizations targeted by relaxation. But before doing that, we need to define some basic concepts we are going to refer to during this section.

4.4.2 Definitions

Notation:

- $Vars$ is the set of all register, memory and queue state variables in the circuit specification.
- $Versions$ is a set of integers.
- $Vals$ is the set of all values that the state variables in $Vars$ can take.
- The set S of states $s \in S$ is the set of functions $S = Vars \rightarrow Vals$.
- We write (v, j) to denote the state variable $v \in Vars$ with version $j \in Versions$.
- C is the set of all boolean expressions in our specification language. C^R is the set of all boolean expressions in which state variables have versions associated with.

- We write $CNF(c)$ to denote the *conjunctive normal form* of a boolean expression $c \in C$.
- We write $P[e/d]$ for the predicate P with the expression d replaced by another expression e.

Definition 1: *Relaxation is the process of replacing each version of each state variable in all the preconditions of the rules with its earliest safe version.*

Definition 2: *Given the specification derived in the first step of the algorithm and some version* `j` *of a state variable, we call* `k` *an earlier version of* `j` *if* `k<j`.

Definition 3: *Given a rule R_l with precondition $P_l \in C^R$ which reads (v, j) and given an earlier version (v, k) of (v, j), we say that (v, k) is safe for (v, j) if the following property holds:*

> *If P_l is true with (v, j) replaced by (v, k), then it is also true with (v, j), i.e. $P_l[(v, k)/(v, j)]$ implies P_l.*

This is an application of the following more general rule: Assume a predicate $P[e/d]$ implies P. Then for any rule with precondition P, we can (subject to liveness concerns) use the predicate $P[e/d]$ instead of P.

Definition 4: *We define $f : C \times S \rightarrow Bool$ to be the function that takes a boolean expression $c \in C$ and a state $s \in S$ and returns the boolean value that c evaluates to in state s of the circuit.*

Definition 5: *Two boolean value expressions are mutually exclusive if they cannot simultaneously be true. More formally, if c_1 and c_2 are two boolean value expressions in C, then*

$$(c_1, c_2)_{ME} \text{ iff } f(c_1, s) \wedge f(c_2, s) = \mathsf{false}, \forall s \in S$$

The mutual exclusion test is implemented using resolution [10] and a set of reduction and simplification rules. We discuss how we implemented this test in detail in Section 4.6.2.

Definition 6: For $c \in C$, let $CNF(c) = \wedge_{i \in I}(\vee_{j \in J} T_{ij})$ be its conjunctive normal form. *We define the set of valid relaxable expressions* $VRE(c)$ *as follows:* $VRE(c) = \wedge_i(\vee_j T_{ij}) \cup \{\vee_j T_{ij}, \forall i \in I\} \cup \{T_{ij}, \forall i \in I, \forall j \in J\}$.

The idea behind the set of valid relaxable expressions is the following. Our relaxation algorithm includes a step which tries to prove that an expression c_1 logically implies another expression c_2, i.e. $c_1 \rightarrow c_2$. By transforming an expression c into conjunctive normal form, we make sure that all the negation operations will only be applied to symbols rather than other expressions. We know that implication is preserved under conjunction and disjunction: if $A \rightarrow B$ and $C \rightarrow D$, then $A \wedge C \rightarrow B \wedge D$ and also $A \vee C \rightarrow B \vee D$. This nevertheless does not hold for negation: $A \rightarrow B$ does not imply $\tilde{}A \rightarrow \tilde{}B$. Transforming first c_1 and c_2 into CN-form ensures that once we prove implication for simpler expressions in $VRE(c_1)$ and $VRE(c_2)$, we can repeatedly compose these expressions to automatically obtain a proof that $c_1 \rightarrow c_2$. The automatic proof holds because the only operation that cannot correctly be applied is negation; but in CNF, all the negation operations exist only within T_{ij} and therefore will not be applied to derive an expression from its set of valid relaxable expressions.

4.4.3 The Algorithm

The algorithm processes all the rules in reverse order, repeatedly attempting to replace the current version of each variable in the enabling condition of the rule, with the previous corresponding version, starting from the immediately preceding rule. For rule R_l, Fig. 4.3 shows how to obtain a new expression from `Exp` in $VRE(P_l)$ by replacing (v, j) in `Exp` with its earliest safe version. $k-1$ is the current version of the state variable v that a replacement is tried with and k initially starts at $k = j$. A replacement is successful if either of the following three conditions is

true:

- The currently processed rule does not update state variable v (`Case 1`).
- The condition of the rule that computes the earlier version and the current enabling condition are mutually exclusive (`Case 2`).
- The expression `Exp` with the earlier version instead of the current one implies the expression `Exp` with the current version (`Case 3`).

```
replace ((v,j),Exp,k)
if k = 0
    then Exp
else if R_k does not update v    (Case 1)
         then replace((v,j),Exp,k-1)
     else if (P_k,P_l) mutual exclusive    (Case 2)
            then replace((v,k-1),Exp[(v,k-1)/(v,j)],k-1)
          else if Exp[(v,k-1)/(v,j)]  implies  Exp    (Case 3)
                 then replace((v,k-1),Exp[(v,k-1)/(v,j)],k-1)
               else Exp
```

Figure 4.3: Relaxation Algorithm

Case 3: In some cases, our algorithm needs to check whether `Exp[`$(v, k-1)$`/`(v, j)`]` implies `Exp` or not. The implication proof is implemented using resolution and a set of reduction and simplification rules. In the general case, the search space for the resolution technique is exponential in the number of formulas describing the problem. Therefore, there is a trade-off in how we choose `Exp` in Fig. 4.3 to efficiently decide whether for a rule, the enabling condition with the earlier version instead of the current one implies the enabling condition with the current version. If we choose `Exp` to be $CNF(P_l)$, we only need to prove at most one implication per earlier version of each state variable in P_l. The disadvantage is that the expressions involved will be more complex than any other expression

in $VRE(P_l)$ and the resolution mechanism will reflect this increased complexity. If, on the other hand, we choose shorter expressions in $VRE(P_l)$, the search space will be greatly reduced. We can then just compose simpler implication relations using conjunction and disjunction to prove `Exp[`$(v, k-1)$`/`(v, j)`]` $\rightarrow$ `Exp`. The disadvantage of this approach is that, while it is true that if $A \rightarrow B$ and $C \rightarrow D$, then $A \wedge C \rightarrow B \wedge D$ and also $A \vee C \rightarrow B \vee D$, the reverse is not necessarily true. This means that simpler composing implications might be false while the more complex implication is still true.

The relaxation algorithm first tries to prove $T_{ij}[(v, k-1)/(v, j)] \rightarrow T_{ij}, \forall i \in I, \forall j \in J$. If there is at least one T_{kl} for which the corresponding implication cannot be proved to be true, the algorithm tries to prove a weaker implication relation for an expression in $VRE(P_l)$ containing each such T_{kl}. The algorithm works its way through $VRE(P_l)$ in the same manner until an implication is true that contains T_{kl}. If the checks for all expressions in $VRE(P_l)$ containing T_{kl} failed, including the top-most one (i.e. $P_l[(v, k-1)/(v, j)] \rightarrow P_l$), the algorithm concludes that $(v, k-1)$ cannot be replaced for v_k in P_l.

The implication test is a simple call of `ResolRedSimpl` in Section 4.6.2, where the `currentClauseSet` is obtained as seen in Fig. 4.4, with `EC`$_j \in C$ the enabling condition `Exp`, (v, r) the state variable in the original numbering and (v, i) the state variable in the transformed numbering.

Case 2: The relaxation algorithm may also test whether P_k and P_l are mutually exclusive. Similar to the implication test, mutual exclusion testing is implemented using the same combination of resolution and simplification/reduction rules.

Assume we want to test whether $\mathtt{EC}_j(x,i) \rightarrow \mathtt{EC}_j(x,r)$, where

(x,i) = {earlier version of (x,r) |

$\nexists$1>i (x,l) = earlier version of (x,r)}.

Let $\mathtt{R}_k$ be the rule that updates (x,r) with $\mathtt{EXPR}(x,i)$.

This implies:

(x,r) = if $\mathtt{P}_k(x,i)$ then $\mathtt{EXPR}(x,i)$ else (x,i).

This translates to proving the following:

$$\mathtt{EC}_j(x,i) \rightarrow \mathtt{P}_k(x,i) \wedge \mathtt{EC}_j(\mathtt{EXPR}(x,i)) \vee \tilde{}\,\mathtt{P}_k(x,i) \wedge \mathtt{EC}_j(x,i),$$

which is equivalent to proving:

$$\mathtt{EC}_j(x,i) \wedge \tilde{}\,(\mathtt{P}_k(x,i) \wedge \mathtt{EC}_j(\mathtt{EXPR}(x,i))) \wedge \tilde{}\,(\tilde{}\,\mathtt{P}_k(x,i) \wedge \mathtt{EC}_j(x,i)) \rightarrow \mathsf{false}$$

$$\mathtt{EC}_j(x,i) \wedge (\tilde{}\,\mathtt{P}_k(x,i) \vee \tilde{}\,\mathtt{EC}_j(\mathtt{EXPR}(x,i))) \wedge (\mathtt{P}_k(x,i) \vee \tilde{}\,\mathtt{EC}_j(x,i)) \rightarrow \mathsf{false}$$

$$(\mathtt{EC}_j(x,i) \wedge \mathtt{P}_k(x,i) \vee \mathtt{EC}_j(x,i) \wedge \tilde{}\,\mathtt{EC}_j(x,i)) \wedge (\tilde{}\,\mathtt{P}_k(x,i) \vee \tilde{}\,\mathtt{EC}_j(\mathtt{EXPR}(x,i))) \rightarrow \mathsf{false}$$

But $\mathtt{EC}_j(x,i) \wedge \tilde{}\,\mathtt{EC}_j(x,i)$ false =>

$$\mathtt{EC}_j(x,i) \wedge \mathtt{P}_k(x,i) \wedge (\tilde{}\,\mathtt{P}_k(x,i) \vee \tilde{}\,\mathtt{EC}_j(\mathtt{EXPR}(x,i))) \rightarrow \mathsf{false}$$

But $\mathtt{P}_k(x,i) \wedge \tilde{}\,\mathtt{P}_k(x,i)$ false, which implies

$$\mathtt{EC}_j(x,i) \wedge \mathtt{P}_k(x,i) \wedge \tilde{}\,\mathtt{EC}_j(\mathtt{EXPR}(x,i)) \rightarrow \mathsf{false}$$

The set of clauses input to the resolution-based mechanism is:

1. $\mathtt{EC}_j(x,i)$
2. $\mathtt{P}_k(x,i)$
3. $\tilde{}\,\mathtt{EC}_j(\mathtt{EXPR}(x,i))$

We call this set $\mathtt{CCS}((v,r),(v,i),\mathtt{EC}_j)$.

Figure 4.4: Obtaining the Set of Clauses Input to the Resolution-based Mechanism

4.4.4 Relevance

The relaxation algorithm is especially well suited for use with queues. Inserting an element at the tail of a queue does not affect the element that was at the head of the queue before the insertion. Rules that test the first element of a queue remain enabled regardless of the number of elements inserted at the tail of the queue, provided that no rule previously removes the head of the queue. This property allows such a rule to test the initial version of the queue, rather than versions produced by earlier rules.

Conceptually, the algorithm could include an initial phase that can in many cases order the rules so as to match the flow of data in the pipeline. Being able to put the rules in this order is sufficient (but not necessary) to ensure that they all test the initial version of each queue.

4.4.5 Simple Example

Figure 4.5 presents the resulting specification after relaxing the description obtained in Figure 4.2. To increase the readability of the examples, we will use v_i for (v, i). As a result of relaxation, the pipeline stages execute in parallel rather then sequentially.

Let's go step by step through the process of relaxing the enabling condition of rule 3: `<JRZ r l> = head(iq2) and notin(rq2,<INC r _>)`. To correctly replace $\texttt{iq}_2$ and $\texttt{rq}_2$ in 3 with $\texttt{iq}_0$ and $\texttt{rq}_0$, we follow a two-step proof.

- **Step 1:**

  `<JRZ r l> = head(iq1) and notin(rq1, <INC r _>)` $\rightarrow$
  `<JRZ r l> = head(iq2) and notin(rq2, <INC r _>)`

  Proof: If `<JRZ r l> = head(iq1)` is true, then `<INC r> = head(iq1)` is false. Therefore rule 2 cannot execute and $\texttt{rq}_2 = \texttt{rq}_1$, $\texttt{iq}_2 = \texttt{iq}_1$.

```
// Instruction Fetch Stage
1:  true -> iq1 = append(iq0,im[pc0]), pc1 = pc0+1;
// Register Operand Fetch Stage
2:  <INC r> = head(iq0) and notin(rq0, <INC r _>) ->
    iq2 = tail(iq1), rq2 = append(rq1, <INC r rf1[r]>);
3:  <JRZ r l> = head(iq0) and notin(rq0, <INC r _>) ->
    iq3 = tail(iq2), rq3 = append(rq2, <INC rf2[r] l>);
// Compute and Writeback Stage
4:  <INC r v> = head(rq0) ->
    rf4 = rf3[r->v+1], rq4 = tail(rq3);
5:  <JRZ v l> = head(rq0) and v = 0 ->
    pc5 = l; iq5 = nil, rq5 = nil;
6:  <JRZ v l> = head(rq0) and !(v = 0) ->
    rq6 = tail(rq0);
```

Figure 4.5: Specification After Relaxation

- **Step 2:**

```
<JRZ r l> = head(iq0) and notin(rq0, <INC r _>) →
<JRZ r l> = head(iq1) and notin(rq1, <INC r _>)
```

Proof:

```
iq1 = if true (i.e.  rule 1 executes)
          then append(iq0,im[pc0])
       else iq0
```

which implies that `head(iq1) = head(iq0)` if `iq0` is not empty. But `<JRZ r l> = head(iq0)` and therefore `iq0` is not empty. Therefore `<JRZ r l> = head(iq0)` → `<JRZ r l> = head(iq1)` Also $rq_1 = rq_0$ implies `notin(rq0, <INC r _>)` → `notin(rq1, <INC r _>)`.

All the precondition relaxations follow the same kind of proof.

4.4.6 Correctness

This section gives an informal proof of correctness for the relaxation algorithm. The detailed formal proof is given in Chapter 5. Correctness implies two concepts: simulation (or safety) and non-termination (or liveness). Simulation says that the resulting system after applying some transformation never does anything that the system before the transformation could not do. Non-termination says that the transformation does not stop the progress of the system's execution.

- **Simulation**

  We want to prove that if the enabling condition of rule R_l is true after relaxation, then the condition would also be true before relaxation. This implies that if a rule in the transformed numbering executes, the rule would also execute in the original numbering and yield the same result.

  $$P_l[(v,k)/(v,l-1)] \texttt{ = true => } P_l(v,l-1) \texttt{ = true}$$

  Proposition: *For any expression* ***Exp*** *in* $VRE(P_l)$ *that was successfully relaxed by our algorithm,* ***Exp*** $\texttt{[}(v,n)/(v,q)\texttt{]} \rightarrow$ ***Exp*** $\texttt{(}(v,q)\texttt{)}$*, where* (v,n) *is any earlier version that successfully relaxed* (v,q) *in* ***Exp****.*

  Proof:

  We will present a proof by induction.

  Basis: If (v,n) is the previous earlier version of (v,q), then either:

  - `Exp[`$(v,n)/(v,q)$`]` $\rightarrow$ `Exp[`(v,q)`]` or
  - $(P_l, P_{n+1})_{ME}$

  In the first case, the proposition is immediately true. In the second case $(v,n) = (v,q)$ `=> Exp[`$(v,n)/(v,q)$`] = Exp(`(v,q)`)`.

  Induction Step: Assume `Exp[`$(v,t)/(v,q)$`] = Exp(`(v,q)`)`, where (v,t) is the k-th earlier version that successfully relaxed (v,q) in `Exp`. We want to prove that `Exp[`$(v,r)/(v,q)$`] = Exp(`(v,q)`)`, where (v,r) is the (k+1)-th earlier version that successfully relaxed (v,q) in `Exp`.

  If (v,r) and (v,t) are the (k+1)-th and k-th earlier versions that suc-

cessfully relaxed (v, q), then (v, r) is the earlier version of (v, t) that successfully relaxed (v, q) in `Exp`. This implies that either:

- `A: Exp[`$(v, r)/(v, t)$`]` $\rightarrow$ `Exp[`(v, t)`]` or
- `B:` $(P_l, P_{r+1})_{ME}$

If `A = true` then `Exp[`$(v, r)/(v, q)$`]` $\rightarrow$ `Exp[`$(v, t)/(v, q)$`]` $\rightarrow$ `Exp(`(v, q)`)`. This is true because `A` $\rightarrow$ `B` and `B` $\rightarrow$ `C` imply `A` $\rightarrow$ `C`. Therefore `Exp[`$(v, r)/(v, q)$`]` $\rightarrow$ `Exp(`(v, q)`)`.

If `B = true` then $(v, r) = (v, t)$, which implies that `Exp[`$(v, r)/(v, q)$`]` $=$ `Exp[`$(v, t)/(v, q)$`]` $\rightarrow$ `Exp(`(v, q)`)`. This concludes the proof of the induction step. Therefore `Exp[`$(v, n)/(v, q)$`]` $\rightarrow$ `Exp(`(v, q)`)`, where (v, n) is any earlier version that successfully relaxed (v, q) in `Exp`.

QED (Proposition).

Therefore, for any `Exp =` $T_{ij} \mid \vee_{j' \in J} T_{ij} \mid \wedge_{i' \in I} (\vee_j T_{ij})$ successfully relaxed by our algorithm, `Exp[`$(v, k)/(v, l-1)$`]` $\rightarrow$ `Exp(`$(v, l-1)$`)`.

But $P_l = \wedge_i (\vee_j T_{ij})$ and we can combine the implications above using conjunction and disjunction, to derive $P_l[(v, k)/(v, l-1)]$ `= true =>` $P_l(v, l-1)$ `= true`.

- **Non-termination**

  Informally, we want to prove that if from any state in the original numbering there exists a step that can be taken, then there exists a step that can be taken from a state in the transformed numbering. The intuition behind the non-termination proof is that since a rule R in the transformed numbering tests a state previous to the current state, if R is enabled in the original numbering but not in the transformed one, *some* rule does execute in the transformed numbering and modifies the state originally tested by R.

  Proposition: *Assume R_l is the rule that executes in the original numbering, but not in the relaxed one. We want to prove that* $\exists n < l$ `s.t.` *R_n executes in the transformed numbering.*

  Proof:

  We will prove this proposition by contradiction. Let's assume $\nexists n < l$

s.t. R_n executes in the transformed numbering. This means that R_i in the transformed numbering does not execute, $\forall i = 1, 2, ..., l-1$, which implies that $(v, l-1) = (v, l-2) = ... = (v, 1) = (v, 0)$, $\forall v \in Vars$ written by R_i.

But in the original numbering, rule R_l's precondition tests version $l-1$ of the state variables. Following the rationale above, the $l-1$-th version of any state variable is, however, the same as any earlier version of that state variable. Therefore, rule R_l's preconditions before and after relaxation will use the same exact values of the state variables. This means that either both preconditions are true or false, simultaneously. But this is in contradiction with our hypothesis, and therefore, $\exists n < l$ s.t. R_n executes in the transformed numbering.

QED.

4.4.7 Trade-offs

The implementation exposes a trade-off between concurrency and cycle time. We chose to maximize concurrency in our system. By default, if two rules simultaneously evaluate their preconditions to true, they are both going to execute in the current clock cycle, either in parallel if there are no data dependencies or sequentially if there are. Our implementation does not handle multicycle operations. The execution of each rule is atomic and must complete within the duration of a clock cycle. Chapter 7 presents a method for the automatic pipelining of sequential circuits. Our pipelining algorithm works by repeatedly extracting a computation from the critical path, moving it into a new pipeline stage, then using speculation to compute the next value produced by the extracted computation in a clock cycle before its correct, new value can be determined. Nevertheless, the technique we present in Chapter 7 is only concerned with functional pipelining.

4.4.8 Circuit Optimizations

Relaxation is a transformation geared towards optimizing the resulting circuit for throughput and clock cycle time. By eliminating dependencies in the specification, relaxation increases the concurrency, which boosts the throughput of the system. Since after relaxation the execution of the rules is not strictly sequential anymore and preconditions of multiple rules are now evaluated simultaneously, the transformation has the potential to shorten the critical path of the circuit and produce an implementation with a shorter clock cycle.

4.5 Global Scheduling

Starting from the relaxed specification derived in the previous step and given a designer-specified length for each unbounded queue, the synthesis algorithm must generate an implementation that does not exceed that length. Our system is designed to maximize concurrency; the challenge is to schedule the maximum number of rules for execution while ensuring that the queues are within specified length at the end of the clock cycle. Rules can still insert elements into full queues if there is a guarantee that enough following rules will execute in the current cycle and remove elements from them.

The global scheduling algorithm is the key to efficient pipelining; it also may reduce the area of the resulting circuit.

4.5.1 Basic Idea

The scheduler augments each rule that inserts an element into a queue to ensure that it never causes any of the corresponding finite hardware buffers to overflow. The basic approach is to assume all queues are within

length at the beginning of the clock cycle and schedule only those rules for firing that are 1) enabled and 2) whose combined execution leaves the queues within their length at the *end* of the clock cycle. All the other rules remain unchanged. As part of this process, queue insertions are prioritized. In hardware, global scheduling corresponds to generating the control signals for the combinational logic.

The rest of the section presents the scheduling algorithms for both acyclic and cyclic specifications, gives the correctness proof, then discusses the characteristics of the algorithm that boost the throughput of the resulting circuit.

4.5.2 Definitions

Definition 1: *A rule graph is a directed graph* `RG = (N,E)`*, where each node in* `N` *corresponds to a rule in the specification. There is a directed edge between two rules if the first rule inserts items into a given queue and the second rule removes items from the same queue.*

Definition 2: *An appending rule is a rule whose set of updates contains at least one insertion of an element into some queue.*

4.5.3 Acyclic Specifications

An acyclic specification is a specification whose rule graph contains no cycles. The large majority of the circuits have acyclic specifications; feedback loops in a circuit do not create cycles in the rule graph because the register on the feedback edge is not a conceptually unbounded queue in the specification.

Idea

The intuition in the acyclic case is that it is possible to compute, knowing

the initial number of elements in each queue at the beginning of the cycle and considering the rules one at a time, the current number of elements in each of the queues after each rule's execution. Therefore, it is enough if the algorithm processes the rules in order and gradually gathers all the conditions that need to be fulfilled to avoid overflow of the queues.

Algorithm

The additional no-overflow constraints are obtained as shown in Fig. 4.6. We use the following notation:

- $Queues$ is the set of all the queues in the circuit specification.
- R' is rule R augmented with the corresponding additional constraints.
- $room : Queues \times S \rightarrow Int$ is a function that takes a queue $q \in Queues$ and a state $s \in S$ and returns the number of empty locations of q in s.
- $eval : Bool \rightarrow \{0, 1\}$ is a function that returns 1 for true and 0 for false.
- $index(X)$ is a function that takes a set of rules and returns the set of indices of all the rules in X.
- Let U be the set of all possible updates in our specification language. We will use $u_i \in U$ to refer to the update part of some rule R_i.
- Let $g : U \times S \rightarrow S$ be an update function. $g(u, s) = s'$ applies the updates in u to the state s and returns the modified state s'.

The algorithm processes all the rules in topological sort order and for each appending rule `R`, augments its precondition with an additional enabling condition. For the additional condition for a queue `q` to evaluate to true, one of the following alternatives has to be true:

  - There is room in `q` when the rule executes.
  - Future rules will execute in the current cycle and remove items from `q` to make room for the result of `R`.

Therefore, the algorithm considers both queue removals and previous insertions to augment the enabling condition of each rule so that it does not execute if it would overflow the queue.

```
for each rule R_i [in topological sort order]
    Q = {q | R_i inserts into q}
    if Q != nil
    then for each q ∈ Q
        I = {R_j | R_j inserts into q}
        D = {R_k | R_k removes from q}
        Select(R_i, q) =
            if (R_i is the only rule in I) or
             ∀i1, i2 ∈ I, (i1, i2) mutually exclusive
            then "∀k ∈ index(D).                          (Case 1)
                 room(q, s_0) + Σeval(f(P'_k, s_k)) > 0"
            else "∀k ∈ index(D).∀j ∈ index(I).j < i.      (Case 2)
                 room(q, s_0) + Σeval(f(P'_k, s_k)) > Σeval(f(P'_j, s_j))"
    else NOP
```

Figure 4.6: Computing the Additional Constraints for a Rule in an Acyclic Specification

`Case 1`: The simplest scenario is when there is only one rule (R_i) that inserts elements into queue q or all the rules that insert into q are mutually exclusive. In this case, the only operations of interest with regard to q are entry removals. The additional enabling condition for R_i is $\forall k \in index(D), room(q, s_0) + \Sigma eval(f(P'_k, s_k)) > 0$, where s_0 is the state at the begining of the clock cycle and s_i is defined by the reccursive definition $s_{i+1} = f(P'_i, s_i)\ ?\ g(u_i, s_i)\ :\ s_i$. The condition reads as follows:

> The number of empty locations in `q` at the begining of the cycle plus the number of removals in that cycle (both previous and following R_i) has to be at least 1 for R_i to be allowed to execute and insert a new element into q.

`Case 2`: If there are at least two non-mutually exclusive rules that insert elements into queue q, besides the rules that perform entry removals from q, we have to also consider previous insertions into q, as follows:

The number of empty locations in `q` at the begining of the cycle plus the number of removals in that cycle (both previous and following R_i) has to be at least one larger than the number of previous insertions into `q` for R_i to be allowed to execute and insert a new element into q.

The additional enabling condition for rule R_i inserting into q is:

$$\forall k \in index(D), \forall j \in index(I), j < i,$$
$$room(q, s_0) + \Sigma eval(f(P'_k, s_k)) > \Sigma eval(f(P'_j, s_j))$$

In the actual implementation,

$$\forall k \in index(D), room(q, s_0) + \Sigma eval(f(P'_k, s_k)) > 0$$

is transformed into

$$\forall k \in index(D), (room(q, s_0) > 0) \bigvee eval(f(P'_k, s_k)).$$

For `Case 2`, $\Sigma eval(f(P'_k, s_k)) - \Sigma eval(f(P'_j, s_j))$ is implemented as an INC/DEC unit.

Simple Example

Our simple processor example has an acyclic specification. By applying the algorithm described above for queues of length 1, we obtain the specification in Figure 4.7.

When using unbounded queues, rule number `2`, for example, executes every time the head of the instruction queue $\texttt{iq}_0$ is an `INC` instruction which is not creating a RAW hazard when attempting to read a given location in `rf`. To ensure that queue `rq` of designer-given length will not overflow, rule number `2` also has to check whether one of the following conditions is true, before executing:

- There is one empty slot in `rq` at the beginning of the clock cycle. For our case where the length of `rq` is 1, this is equivalent with saying that `rq` is initially empty.

```
// Instruction Fetch Stage
1:  (<JRZ v l> = head(rq0) and v = 0) or
    Room(iq0) > 0 or
    ((Room(rq0) > 0 or
      <INC r v> = head(rq0) or
      (<JRZ v l> = head(rq0) and !(v = 0))
    and ((<INC r> = head(iq0) and notin(rq0, <INC r _>)) or
        (<JRZ r l> = head(iq0) and notin(rq0, <INC r _>)))) ->
        iq1 = append(iq0,im[pc0]), pc1 = pc0+1;
// Register Operand Fetch Stage
2:  <INC r> = head(iq0) and notin(rq0, <INC r _>) and
    ((<JRZ v l> = head(rq0) and v = 0) or
    Room(rq0) > 0 or
    <INC r v> = head(rq0) or
    (<JRZ v l> = head(rq0) and !(v = 0))) ->
        iq2 = tail(iq1), rq2 = append(rq1, <INC r rf1[r]>);
3:  <JRZ r l> = head(iq0) and notin(rq0, <INC r _>) and
    ((<JRZ v l> = head(rq0) and v = 0) or
    Room(rq0) > 0 or
    <INC r v> = head(rq0) or
    (<JRZ v l> = head(rq0) and !(v = 0)))->
        iq3 = tail(iq2), rq3 = append(rq2, <INC rf2[r] l>);
// Compute and Writeback Stage
4:  <INC r v> = head(rq0) ->
        rf4 = rf3[r->v+1], rq4 = tail(rq3);
5:  <JRZ v l> = head(rq0) and v = 0 ->
        pc5 = l; iq5 = nil, rq5 = nil;
6:  <JRZ v l> = head(rq0) and !(v = 0) ->
        rq6 = tail(rq5);
```

Figure 4.7: Specification After Global Scheduling

– One of the rules processing elements of `rq` will subsequently execute in the current clock cycle.

If the first condition is true, than there exists space in `rq` and adding a new element at the end of it keeps `rq` strictly within length. Otherwise `rq` is initially full and adding a new element at the end of it would not preserve the overflow invariant on `rq`. But if the second condition is true, the rule that subsequently executes removes an element from the head of `rq` and therefore brings the queue within legal length again. In the same way we can explain the additional enabling conditions for rules number `1` and `3`.

4.5.4 Cyclic Specifications

A cyclic specification is a specification whose rule graph contains at least one cycle. Systems that contain cycles include circuits that implement cryptographic algorithms. Introducing additional enabling conditions as described in the previous section raises the possibility of deadlock. This scenario may happen because for each appending rule `R`, its enabling condition tests the execution of rules both before and after `R` in the specification. This may lead to circularities and therefore to deadlock. For acyclic specifications, this is not an issue because the acyclicity ensures that the queues will eventually drain, enabling rules that were originally suspended for lack of space. But this line of reasoning does not hold for cyclic specifications. The key insight is that the additional enabling conditions need not introduce deadlock if there is a way to coordinate the removals and insertions of elements from all of the queues in the cycle so that the removal of each element leaves room for the insertion of some element behind it. The algorithm for cyclic specifications therefore analyzes groups of rules together to generate a global schedule that allows all of the data in a cycle to move together through the cycle.

Example

We use the example in Fig. 4.8 to illustrate the operation of the algorithm for cyclic specifications. To simplify the presentation, we present the rules by themselves, omitting the module decomposition. We also omit any rules that do not affect the contents of queues `x`, `y` and `z`.

```
var a :  int(8);
var x :  queue(int) =  2 ;
var y :  queue(int) =  3 ;
var z :  queue(int);
0:  t = head(x) ->
        y = append(y,(t+3)&15), x = tail(x);
1:  t = head(y) ->
        x = append(x,(t+5)&15), y = tail(y);
2:  t = head(x) and (t&3 = 0) ->
        z = append(z,t), x = tail(x);
3:  t = head(y) and (t&3 = 0) ->
        z = append(z,t), y = tail(y);
4:  t = head(z) and (t&7 = 0) ->
        a = t, z = tail(z);
// implementation constraints
length(x) = 1;
length(y) = 1;
length(z) = 1;
```

Figure 4.8: Cyclic Example

This example is modeled after a random number generation process that starts with two numbers (2 and 3) and repeatedly adds 3, then 5 to each number, retaining the lower 4 bits after each addition. The computation records the values of the numbers when their bottom 2 bits become 0. In our implementation, each number is stored in a queue, and the designer specifies that each queue has a single entry. Because of the cyclic nature

of the specification, the numbers must move through the queues together — if they attempt to move separately, there is no room in the queues. The synthesis algorithm must therefore schedule the rules involved in the cycle (rules 0 and 1) together to coordinate their queue insertions and removals.

Idea

The key idea is to find, for each rule that inserts an element into a queue q, the maximal sets of rules that have to execute together to preserve the "non-overflow" invariant of q, at the end of each clock cycle. To do this, the algorithm starts from each rule and traverses the rule graph on all possible paths, gathering for each rule that it goes through, the conditions that would let that rule execute. We stop if either a rule is not an appending rule, so will always execute when its initial enabling condition becomes true, or if we already traversed that rule on the current path, so we already considered that the rule executes. Once we reach such a point there's no additional information on that path in the circuit that was not already collected at the first traversal. Nothing needs to be added to yield a correct solution. When all paths reach such points, the set of all rules that have to execute together becomes maximal.

Algorithm

Fig. 4.9 shows the algorithm that produces the additional enabling condition for a rule R_i. We use the following notations:

- $CrtPath$ is the currently explored path and we use it for purposes of termination. This variable is initially empty for each symbolic execution of a rule.
- $newR_k$ is what we derive from R_k after enhancing it with the additional constraints.

– $Select(R_i, q)$ has a form similar to the additional condition from the acyclic case.

```
SymbolicExecution (R_i, CrtPath)
Q = {q | R_i inserts into q}
if Q != nil
then for each q ∈ Q
    I = {R_j | R_j inserts into q}
    D = {R_k | R_k removes from q}
    S = if (R_i is the only rule in I) or
          ∀i1, i2 ∈ I, (i1, i2) mutually exclusive
            then D
            else D ∪ I
    CyclicSelect(R_i, q) = Select(R_i, q)
    for each rule R_k ∈ S
        newCrtPath = CrtPath ∪ R_k
        newR_k = if R_k ∈ CrtPath
            then true
            else SymbolicExecution(R_k, newCrtPath)
        newSelect(R_i, q) = CyclicSelect(R_i, q)[newR_k / R'_k]
        CyclicSelect(R_i, q) = newSelect(R_i, q)
    newR_i = (R_i and CyclicSelect(R_i, q))
    R_i = newR_i
else R_i
```

Figure 4.9: Computing the Additional Constraints for a Rule in a Cyclic Specification

The scheduling algorithm processes each rule in the cyclic specification in turn. The symbolic execution of a rule traverses the rule graph on all possible paths and causes the symbolic execution of all the rules R_k on these paths. For each rule, the additional enabling condition gathers all the scenarios that would let that rule execute. The symbolic execution of some rule R_k terminates if either one of the two scenarios below is true:

– R_k is a non-appending rule and in this case $newR_k = R_k$.

– R_k is a rule previously examined on the current path from R_i. This means we already assumed R_k executes on that path, so there's no need to explore further, therefore $newR_k = \mathsf{true}$.

When all the rules on all the possible paths from R_i have been symbolically executed, the final expression for the additional enabling condition of R_i is complete.

Example Revised

If we apply the algorithm to the cyclic example in Figure 4.8, we obtain the following updated enabling conditions:

$$\begin{aligned}
&\mathtt{P_0'} = \mathtt{P_0} \wedge \ \mathtt{[Room(y)>0} \ \vee \ \mathtt{P_1} \ \vee \ \mathtt{P_3} \wedge \\
&\qquad\qquad \mathtt{[Room(z)>0} \ \wedge \ \mathtt{(P_4} \vee \tilde{}\mathtt{P_2)} \vee \mathtt{P_4} \wedge \tilde{}\mathtt{P_2]]} \\
&\mathtt{P_1'} = \mathtt{P_1} \ \wedge \ \mathtt{[Room(x)>0} \ \vee \ \mathtt{P_0} \ \vee \ \mathtt{P_2} \wedge \mathtt{(Room(z)>0} \ \vee \ \mathtt{P_4)]} \\
&\mathtt{P_2'} = \mathtt{P_2} \ \wedge \ \mathtt{(Room(z)>0} \ \vee \ \mathtt{P_4)} \\
&\mathtt{P_3'} = \mathtt{P_3} \wedge \ \mathtt{[Room(z)>0} \ \wedge \ \mathtt{(P_4} \vee \tilde{}\mathtt{P_2)} \vee \mathtt{P_4} \wedge \tilde{}\mathtt{P_2]} \\
&\mathtt{P_4'} = \mathtt{P_4}
\end{aligned}$$

If all the queues (`x, y, z`) are full, rule `0` executes if either :

- $\mathtt{P_1} = \mathsf{true}$ or
- $\tilde{}\mathtt{P_2} \wedge \mathtt{P_3} \wedge \mathtt{P_4} = \mathsf{true}$.

Similarly, rule `1` executes if either:

- $\mathtt{P_0} = \mathsf{true}$ or
- $\mathtt{P_2} \wedge \mathtt{P_4} = \mathsf{true}$.

Therefore, if $\mathtt{P_4} = \mathsf{false}$ then rule `0` and rule `1` can execute together without causing any of the queues to overflow. This is important because if rule `4` does not execute and queue `z` is full, rules `2` and `3` cannot execute either. If an algorithm is not able to detect that rules `0` and `1` could safely execute together, even when queues `x` and `y` are full, the whole system

reaches a deadlock state. Our algorithm does not introduce deadlock as a result of cycles in the specification.

Correctness

This section gives an informal proof of two properties preserved by the global scheduling algorithm: simulation and non-termination with respect to the extended specification obtained by augmenting the current spcification with designer-given queue lengths. The detailed formal proof is given in Chapter 5.

- **Simulation**

  We want to prove that for any sequence of rule executions after scheduling, we can find the same sequence of rule executions before scheduling.

  Proof:

  By induction on the length of the execution sequence we have:

  Induction Base:

  Initially, all queues are empty, all the other state variables have the same values before and after applying the scheduling transformation, and the rule testing starts from the first rule in textual order.

  Induction Step:

  Assume up to the current step the execution sequences are identical. If the current step is not an appending rule, its enabling condition does not get modified after scheduling and therefore it will evaluate to the same truth value after as before scheduling. If the current step is an appending rule and this rule executes after scheduling, then it would have also executed before it. This is true because the enabling condition is strictly stronger after than before applying the scheduling transformation.

– **Non-termination**

We want to prove that if from any state of the system before scheduling there exists a rule R that can execute while keeping all the queues within their designer-specifed lengths, then there exists some rule R' that can execute in the new system that contains the additional enabling conditions. We also want to prove that such a rule R' does not exist iff any enabled rule R would overflow at least one of the queues.

Proof:

The proof for the first part proceeds by contradiction:

Assume there exists no rule R' that can execute in the new system. This implies that in the new system $\forall q \in Queues, room(q, s_{eoc}) = room(q, s_{crt})$ (1), where s_{eoc} is the state of the system at the end of the clock cycle and s_{crt} is the current state for rule R' derived from R by construction of the new system. From the hypothesis we know that firing R in the initial system keeps all the queues withing their specified lengths at the end of the clock cycle, therefore $room(q, s_{eoc}) \geq 0$ (2).

If R is an appending rule, from (2) we infer that $room(q, s_{crt}) > 0$ (3). We can easily prove that this also holds in the new system. From (1) and (3) we have that in the new system $room(q, s_{eoc}) > 0$ (4). From the construction of the new system we have that if R is an appending rule, than R' is also an appending rule; from this and (4) we have that in the current state of the new syetem, there is space in q for an appending rule (in our case R') to execute (5). Regardless whether R is an appending rule or not, because R executes we infer by construction of the new system that the corresponding rule (in our case R') is also going to evaluate the part of its enabling condition which excludes the overflow test (if any) to true. From this and (5) we have that R' would execute as well,

which contradicts our initial asumption and therefore completes the proof for the first part of the non-termination property.

We now want to prove the second part of the non-termination proof, which says that rule R' does not exist iff enabled rule R would overflow at least one of the queues. Let's call this theorem PART2. Proving PART2 is equivalent to proving that letting R execute will cause some queue in the initial system to overflow iff letting R' execute will cause some (potentially different) queue in the new system to overflow.

To prove this, we start from the current states in the initial and new systems, in which we know that all corresponding queues have the same number of elements. We then only have to prove that corresponding rules in these two systems, following R and R', either both execute or neither does. This is sufficient because it proves strict equality betwen the lengths of corresponding queues in the two systems, at the end of the cycle. Let r and r' be the corresponding rules immediately following R and R' in the initial and new systems. If r and r' are not appending rules, they both execute if their enabling conditions — which are identical — evaluate to true. Otherwise, none of them executes. If r and r' are appending rules, we reduced proving PART2 for R and R' to proving PART2 for r and r'. Because the number of rules following R and R', correspondingly, is finite, we will eventualy reach the last rules in the two systems, where PART2 holds, since there are no more following rules. This result completes the non-termination proof.

– **Correctness for Groups of Rules**

For cyclic specfications, the algorithm considers the coordinated execution of groups of rules, rather than rules in isolation. The simulation theorem for a sequence of rules is virtually identical to the simulation proof for one rule. The non-termination proof goes

by contradiction and works on a sequence of rules instead of a single rule at a time to infer that some rule in the transformed system will have its enabling condition satisfied, and therefore execute.

Trade-offs

Our approach tries to maximize concurrency in the resulting implementation by testing each rule in turn and scheduling as many rules for execution as possible, provided that the queues in the system will not overflow because of it. Though this transformation results in a potential increase of the throughput of the circuit for given maximum queue lengths, its final clock cycle time may also increase. The main source of this potential increase is the fact that now there will be more rules that execute in parallel, and therefore more operations to be performed. Other causes may be that the additional enabling condition (1) tests the truth value of other rule's preconditions, which may not be yet available and (2) contains additional and-or logic that computes its overall truth value. Because generally, the main source of delay is updates in a rule taking longer than concurrent updates in a different rule, we consider that our approach is justified.

Circuit Optimizations

Global scheduling is the enabling technology for efficient pipelining. The key insight is that, every clock cycle, the number of rules that can execute and insert into a queue `q` can be bigger than the number of empty slots in `q`, without causing `q` to overflow. The condition is that enough rules will also execute in that clock cycle and remove elements from `q`, leaving it within length at the end of the clock cycle. Applying this mechanism boosts the throughput of the circuit.

The other advantage of globally scheduling all the operations in the system is that it may reduce the area of the resulting circuit. As we

will comment on in Section 4.9, this is mainly possible because the algorithm can produce a deadlock-free implementation with minimum length hardware buffers, as long as the execution of the original specification is deadlock-free for the required queue lengths.

4.6 Symbolic Execution and Optimizations

Symbolic execution derives an expression for each state variable. This expression computes the value of the variable at the end of each clock cycle in terms of the values of the state variables at the begining of the cycle. These expressions define the operations that the combinational logic component of the circuit performs. The transformations in Section 4.6.2 target the optimization of the combinational logic in the resulting circuit by optimizing the expressions obtained at symbolic execution.

4.6.1 Symbolic Execution

Symbolic execution determines a new value for each state variable at the end of the clock cycle in terms of the values at the start of the clock cycle. It does this by substituting out the intermediate versions of each state variable in the specification obtained in the previous step. The result is an expression, in the original versions of the state variables, for each use of each state variable in the system. The versions at the last rule are latched back into the state variables at the end of the clock cycle, and provide the initial values for the start of the next clock cycle. Since only a subset of the rules may execute in a given clock cycle, the expressions contain conditionals.

4.6.2 Optimizations

To improve the quality of the synthesized circuit, the compiler optimizes the expressions, using common sub-expression elimination (CSE) and mutual exclusion testing. If an expression contains a value that will never actually occur in practice because the conditions required to obtain the value are mutually exclusive, the computation of that value is eliminated from the expression. A typical example is a value obtained if both a `JRZ` and an `INC` instruction are at the head of the instruction queue. Obviously, the instruction must be either a `JRZ` instruction or an `INC` instruction, but not both. So such a value will never be computed in the actual circuit. The mutual exclusion testing is implemented using resolution and simplification and reduction rules.

The resolution method shows whether a theorem logically follows from its axioms; it is a form of proof by contradiction that involves producing new clauses called resolvents, from the union of the axioms and the negated theorem. Resolution is guaranteed to produce a contradiction if the theorem follows from the axioms. The basic rule of inference is the resolution principle:

$$\text{From } (A \vee B) \text{ and } (\tilde{}A \vee C) \text{ infer } (B \vee C).$$

Figure 4.10 shows the reduction and simplification rules that the mutual exclusion test is using. As we can see, a large number of them are description language-specific. In Figure 4.10 we use `t` and $\mathtt{t}'$ to refer to tuples, $\mathtt{e}$, $\mathtt{e}_1$, $\mathtt{e}_2$, $\mathtt{e}'$, $\mathtt{e}'_1$ for expressions and `x` and `y` for integer type variables or tuple fields, or integer numbers. Informally, an expression is one of the following:

- A basic type.
- An array.
- The result of an array update.

– The result of an operation on queues.

– The result of an arithmetic or logic operation on two expressions.

– The result of a conditional expression assignment.

We call two tuples *compatible* if they have the same number of fields and the values of each two corresponding fields are either the same or at least one of them is an _ (*don't care*). Two expressions are compatible if they are the same. Reductions are axioms that take two clauses and produce one new, simpler clause out of them. Simplifications are axioms that rewrite a clause from a more complex to a simpler form. Both reductions and simplifications are written in such a way that speeds up the resolution mechanism.

To prove that the execution of two rules is mutually exclusive, our system must prove that the set of clauses generated by the enabling conditions of the two rules implies false. Figure 4.11 shows the main steps of the mechanism that we use to implement this test. In Figure 4.11, `possibleResolution|possibleReduction|possibleSimplification=` false means that by applying a resolution, reduction or simplification step we obtained the false clause, i.e we successfully proved the implication relation under test. Either of these values being *yes* means that the step was successful, but didn't result in infering the false clause. `currentClauseSet` is the set of clauses that have not yet been used by **ResolRedSimpl**. This set changes as clauses are used to infer other clauses and the newly obtained clauses are added to the existing clause set. To decide whether two rules (in `ruleList`) are mutually exclusive, the mechanism first transforms the enabling conditions of the rules in clausal form — CNF form — then from these it creates the set of clauses that is fed to `ResolRedSimplif`. Each element of this set is one of the disjunctive terms in the clausal form of the enabling conditions of the two rules.

// Set of Reductions:

(t = head(e)) and notin(e,$t^{'}$) = false if t,$t^{'}$ compatible

(t/e_1 = head(e_2)) and ~notin(e_2,t/e_1) = (t/e_1 = head(e_2))

notin(e_2,t/e_1) and ~notin(tail(e_2),t/e_1) = false

(x == y) and (x != y) = false

~(x == y) and ~(x != y) = false

(x == y) and ~(x != y) = (x = y)

~(x == y) and (x != y) = (x != y)

(x < y) and (x > y) = false

~(x < y) and ~(x > y) = (x = y)

(x < y) and ~(x > y) = (x < y)

~(x < y) and (x > y) = (x > y)

(t_1 = e) and (t_2 = e) = false if t_1,t_2 incompatible

notin(e_2,t/e_1) and notin(tail(e_2),t/e_1) = notin(e_2,t/e_1)

~notin(e_2,t/e_1) and notin(tail(e_2),t/e_1) =
(t/e_1 = head(e_2)) and notin(tail(e_2),t/e_1)

// Set of Simplifications:

notin(null,t/e) = true

notin(e,head(e)) = false

notin(append(e_2,$t^{'}$/$e_1^{'}$),t/e_1) = false if t/e_1,$t^{'}$/$e_1^{'}$ compatible

head(append*(e,t/$e^{'}$)) = head(e) if e != null

notin(append(e_2,$t^{'}$/$e_1^{'}$),t/e_1) = notin(e_2,t/e_1)
if t/e_1,$t^{'}$/$e_1^{'}$ incompatible

notin(tail(e_2),t/e_1) = notin(e_2,t/e_1) if t/e_1 != head(e_2)

append(null,t/e) = [t/e]

append(tail(e_1),t/e_2) = tail(append(e_1,t/e_2))

head*(e) = head(e)

tail(null) = null

tail(head(e)) = null

Figure 4.10: Reduction and Simplification Rules

ResolRedSimpl (currentClauseSet)

```
while {∃ cl1,cl2 ∈ currentClauseSet | possibleResolution = yes}
    newClauseSet = Resolution (cl1, cl2, currentClauseSet)
    currentClauseSet = newClauseSet
if possibleResolution = false
    then return SUCCESS
else if {∃ cl1',cl2' ∈ currentClauseSet | possibleReduction = yes}
        then newClauseSet = Reduction(cl1',cl2',currentClauseSet)
             ResolRedSimpl (newClauseSet)
     else if possibleReduction = false
             then return SUCCESS
          else if {∃ cl ∈ currentClauseSet |
                                    possibleSimplification = yes}
                  then newClauseSet =
                                    Simplify(cl,currentClauseSet)
                       ResolRedSimpl (newClauseSet)
               else if possibleSimplification = false
                       then return SUCCESS
                    else if noPreviousInstance of ResolRedSimpl
                            then return INSUCCESS
                         else backtrack
```

MutualExclusion (ruleList)

```
ResolRedSimpl (formClauseSet (clausalForm (ruleList)))
```

Figure 4.11: Resolution-based Mechanism for Inferring a Mutual Exclusion Relation Between Two Rules

Our mechanism is resolution-based and interleaves resolution-style resolve steps with steps that apply simplification and/or reduction rules specific to our description language to infer the false clause. Although resolution is complete[3], it can be extremely time consuming. In the general case, the search space for the resolution technique is exponential in the number of formulas describing the problem. Fortunately, using the additional reduction and simplification rules while trying to infer mutual exclusion relations makes our mechanism work well in practice.

Any sequential circuit contains combinational logic and memory elements (registers and memories). For some circuit specification, the expressions that our synthesis system obtains after symbolic execution mirror the operations that the combinational logic component of the circuit performs. The type of transformations in this step target optimizations of the combinational logic component. The main result of CSE is reducing combinational logic area by avoiding useless hardware duplication. The potential disadvantage is that after applying CSE for an expression, the wires routing the result of the operation to its use places are longer than if the value of the expression was recomputed where needed. This might have a negative effect on the final clock cycle of the circuit after floorplanning. Mutual exclusion testing targets the clock cycle of the resulting implementation; by eliminating false paths in the circuit and also potentially shortening the wire lengths, it determines a decrease in cycle time. Additional optimizations for the combinational logic can be addressed in the synthesis phase from Verilog down to gate level.

4.6.3 Simple Example

For the example in Figure 2.2, Figure 4.12 shows the expressions producing the values of the last versions for all the state variables.

[3]for first-order predicate logic.

```
let
  t1 = <INC r> = head(iq0) and notin(rq0, <INC r _>)
  t2 = <JRZ r l> = head(iq0) and notin(rq0, <INC r _>)
  t3 = append(iq0, im0[pc0])
  t4 = tail(t3)
  t5 = <INC r v> = head(rq0)
  t6 = <JRZ v l> = head(rq0) and !(v = 0)
  t7 = <JRZ v l> = head(rq0) and v = 0
  t8 = rf0[r->v+1]
iq6 =
  if t7 then nil
  else if t1 then t4
  else if t2 then t4
  else if Room(iq0) > 0 then t3
  else iq0
rq6 =
  if t7 then nil
  else if t5 or t6 then
      if t2 then append(rq0, <JRZ rf2[r] l>)
      else if t1 then append(rq0, <INC r rf1[r]>)
  else rq0
pc6 =
  if t7 then l
  else if Room(iq0) > 0 or t1 or t2 then pc0 + 1
  else pc0
rf6 =
  if t5 then t8
  else rf0
```

Figure 4.12: Results of Symbolic Execution for Simple Example

We illustrate the symbolic execution and optimization principle by discussing the final value of the instruction queue `iq`. If there is a taken branch, the instruction queue is cleared. If there is already an instruction at the head of the instruction queue that can go through the register fetch stage, the final result is obtained by inserting the new instruction into the tail of the queue and removing the instruction from the head of the queue. Otherwise, the circuit checks to see if there is an empty entry in the instruction queue. If so, it fetches another instruction; if not, the instruction queue does not change. Note the introduction of the temporary variables `t1`, `t2`, `t3`, and `t4`. These variables will turn directly into combinational logic in the final implementation of the circuit.

4.7 Verilog Generation

The final step is to generate synthesizable Verilog for the optimized final expressions obtained in the previous phase. These expressions denote the values written back into each state variable at the end of the clock cycle.

Our approach implements *arrayType* variables such as the instruction memory in our simple example as synchronous memory arrays of the specified size. The memories can have more than one write port if otherwise independent operations in the circuit can write different locations of the same array at the same clock cycle. Memories in Verilog are defined using the register declaration. For a 64 word array, each word of 32 bit size we write:

```
reg [31:0] m [0:63];
```

*queueType*s are implemented as a number of registers equal to the corresponding specified length of each queue. For a 32-bit wide queue `q` with

physical length of 2, our algorithm generates the following declaration in Verilog

```
reg [31:0] q [1:0];
```

basicType variables are implemented as hardware registers. In our simple example, the program counter is declared as below

```
reg [31:0] pc;
```

For each state variable, the corresponding expression at the end of the clock cycle defines the combinational logic that computes the new value of the variable at the next clock cycle. Given the state variables and the expressions as produced by the optimized symbolic execution, the Verilog generation is quite straightforward. Each operation in an expression gets mapped to either a library primitive or to one of our Verilog modules that implement the pattern matching mechanism or operations on variables of type *arrayType* and *queueType*. The value of the final expression for each state variable feeds back into the corresponding register, memory or FIFO queue.

4.7.1 Trade-offs

Our algorithm is geared towards maximizing concurrency in the resulting circuit. Considering this goal, we make the simplifying assumption that as many resources are available as necessary. Therefore, our algorithm generates dedicated hardware components for each operation. While this assumption exploits all the hardware concurrency available in the circuit, the resulting implementation may have prohibitive area or power requirements. If this is a problem in practice, resource sharing can be implemented during synthesis from the Verilog implementation that our algorithm produces.

4.8 Target Applications

Our approach targets the class of circuits that are naturally described as a composition of interacting sub-systems. An application that needs to make sure that certain operations happen at precisely specified times is not a good candidate for our automated synthesis system. This is a consequence of the lack of a timing constraint mechanism in the current implementation. When specifying very efficient protocols, the designer may need to tightly synchronize the execution of different parts of the circuit or explicitly specify that some operations happen in parallel. The parallel execution of two operations can be expressed as a zero-latency constraint between those operations. Therefore the absence of an explicitly parallel abstract execution model in the framework does not impose additional limitations besides the ones incurred by the absence of a built-in timing constraint mechanism in the implementation. To make it possible to synthesize efficient circuits whose specifications include timing constraints, our system needs to interface with another component that implements this constraint mechanism. Examples of good candidate applications for efficient automated synthesis using our system are DSPs, embedded systems and pipelined processors. If the goal of the designer is (1) prototyping for different hardware platforms or (2) solution space exploration in search of a good implementation as basis for further manual optimization, the application domain is even larger.

4.9 Synchronous vs Asynchronous Logic

Targeting circuits implemented as asynchronous logic may be an interesting alternate output option for our synthesis algorithm because of the potentially important advantages of asynchronous over their synchronous counterparts: no clock skew worries, lower power consumption,

average-case rather than worst-case performance, better technology migration potential. There is no fundamental reason we can't generate asynchronous circuitry, but instead we chose to generate synchronous implementations because of the following reasons:

- Synchronous circuit design is more widely used than asynchronous design. We want our designs to be able to interoperate with other parts of a system, which are more probably implemented as synchronous logic. We also want to make it possible to use existing methodologies to further optimize the resulting circuit. Most of the current methodologies are not able to support asynchronous designs.
- Asynchronous design raises challenging problems that do not appear in synchronous design. One of the most important ones is the completion detection problem. Signaling the completion of a pipeline stage or operation of a functional unit requires extra time, thus increasing the theoretical average-case delay. Another reason is that asynchronous circuits need non-multiplexed multi-ported memories, which get quickly more expensive in both area and propagation delay with the number of ports. Since there is no clock to synchronize with, multiplexing ports is usually done by implementing some kind of a synchronization protocol for the memory accesses, which defeats the point of an asynchronous implementation.

Adapting our current approach to automatically generate asynchronous rather that synchronous circuitry constitutes interesting future work.

4.10 Summary

The synthesis algorithm is the enabling technology for generating a parallel, synchronous implementation starting from a high-level, modular,

sequential and asynchronous specification. The idea is to automatically compose the module definitions to generate a global schedule for all the operations in the rules.

Our implementation goal is to maximize concurrency by maximizing the number of rules that can execute at the same clock cycle. As a result, the throughput of the pipeline increases, which in turn increases the concurrency of the generated circuit. The downside is that allowing more rules to execute may also lead to an increase in clock cycle time. The step in our synthesis algorithm that significantly reduces the cycle time is applying the relaxation algorithm. This technique also increases the parallelism of the resulting implementation. Mutual exclusion testing is the other step that reduces the clock cycle time, mainly by discovering and eliminating the false paths in the circuit.

To further increase the concurrency of the resulting circuit, we adopt a non-resource sharing policy. Operations that would otherwise have structural hazards because of using the same resources can execute in parallel, using dedicated hardware components. This approach trades circuit area for concurrency and therefore it may incur a potentially substantial increase in silicon area. Our system applies techniques like CSE to reduce the combinational logic area by avoiding unnecessary hardware duplication; it also minimizes the feasible lengths of the hardware buffers as a result of deriving a synchronous implementation from the initial asynchronous specification. If silicon area is an important performance metric, the designer can perform resource sharing during synthesis from the Verilog implementation derived in the last step of our algorithm.

Our framework does not provide support for structural pipelining unless it is directly expressed in the specification. The algorithm is based on the underlying assumption that each individual operation takes less time than the clock cycle time to complete. Chapter 7 presents an automated

technique that implements functional pipelining for sequential circuits. The synthesis algorithm generates combinational logic for each operation required to compute the optimized expressions derived at symbolic execution. The existing synthesis algorithm would need to be extended if certain individual basic operations are too expensive to implement in combinational logic.

Our approach targets the class of circuits that are naturally described as a composition of interacting sub-systems and whose behavior does not require the specification of (1) timing constraints or (2) explicitly parallel operations in the circuit. The main candidates are DSPs, embedded systems and pipelined processors.

Chapter 5

Formalism

This chapter contains detailed correctness proofs for the relaxation and global scheduling algorithms. We begin by describing the general formal framework, then for each of these algorithms, we define the specification of a circuit before and after applying the algorithm as direct or extended instances of the general framework. We next state and prove lemmas and theorems regarding the simulation and non-termination properties of our algorithms.

5.1 Formal Definitions

We define a **system** to be a tuple $System\ G = \langle T, ex, f, g \rangle$.

A **transition system** is a set of **transitions** $T = \{t_i \equiv l_i : c_i \Rightarrow u_i\}$.

A **transition** $t \equiv l : c \Rightarrow u \in T$ has a label $l \in Label = \{1, ..., |T|\}$, a condition $c \in C$ and an update $u \in U$.

We have an **external** function $ex : t \rightarrow Bool$ s.t. $ex(t)$ is true if the transition t is observable from the exterior and false otherwise. We say that a transition $t \equiv l : c \Rightarrow u$ is **external** iff $ex(t)$ is true and **internal** otherwise.

$Queues$ is the set of all queues in the circuit specification:

$$Queues(G) = \{queue_k \mid k = 1, ..., maxQueue\},\ queue_k = \{SEQ\ Int\}$$

$Vars$ is the set of all register, memory state variables and queues in the circuit specification.

$$Vars(G) =$$
$$\{reg_i \mid i = 1, ..., maxReg\} \cup \{mem_j \mid j = 1, ..., maxMem\} \cup Queues$$

$Vals$ is the set of all values that the state variables in $Vars$ can take.

A **state** s is a function $s : Vars \rightarrow Vals$.

The **set** S **of states** $s \in S$ is the set of functions $S = Vars \rightarrow Vals$.

We define two functions f and g as follows:

- We assume a set of conditions C, a set of expressions E and a set of updates U. The evaluation of a condition $c \in C$ in some state returns a $Bool$ value. The evaluation of an expression $e \in E$ in some state returns a value in $Vals$. To make the notation more concise, we extend $Vals$ to include $Bool$. The evaluation function $f : E \times S \rightarrow Vals$ returns the value of expression $e \in E$, in state $s \in S$.
- An update function $g : U \times S \rightarrow S$. $g(u, s) = s'$ applies the updates in u to the state s and returns the modified state s'.

Each transition system T defines a **transition relation** $R \subseteq S \times T \times S$.

$$R = \{\langle s, t, s' \rangle.\ t \equiv l : c \rightarrow u \in T \wedge f(c, s) \wedge g(u, s) = s'\}$$

Assume there exists an **initial state** s_0 of T. In s_0 the following are true:

- $\forall v_k \in Vars - Queues$, $s_0(v_k) = initialVal_k$, where $initialVal_k \in Vals$ are the initial values of the registers and memories in the circuit.
- All the queues in the circuit specification are empty.
  $\forall k$ s.t. $1 \leq k \leq maxQueue,\ s_0(queue_k) = \{\}$

An **execution fragment** is a finite alternating sequence of states and transitions.

$$frag = \{s_1 t_1 s_2 t_2 s_3 ... s_n \mid s_i \in S \ . \ t_i \in T \ . \ \langle s_i, t_i, s_{i+1} \rangle \in R\}$$

An **execution** is an execution fragment starting in the initial state s_0.

A state s is **reachable** if s is the final state of some finite execution.

An **execution sequence** is the sequence of states in an execution obtained after dropping the intermediate transitions:

$$\tau = \{s_1, s_2, ..., s_n \mid s_1 t_1 s_2 t_2 ... s_n \text{ is an execution}\}$$

5.2 SPEC

SPEC is an instance of $System$ of the form $\langle T^S, ex^S, f^S, g^S \rangle$. T^S is a transition system that represents the nondeterministic specification of the circuit.

We define a set of expressions E^S in SPEC, where an expression $e^S \in E^S$ is either a constant c (including true and false), a variable v (boolean or not) or an operation on subexpressions:

$e^S ::= c \mid v \mid \rho(e^S{}_1, ..., e^S{}_n)$

Function f^S: $E^S \times S^S \rightarrow Vals$ evaluates expressions of type e^S in some state s^S as follows:

$f^S(c, s^S) = c$

$f^S(v, s^S) = s^S(v)$

$f^S(\rho(e^S{}_1, ..., e^S{}_n), s^S) = \rho(f^S(e^S{}_1, s^S), ..., f^S(e^S{}_n, s^S))$

We also define a simplified set of updates U^S in SPEC

$u^S ::= \mathrm{v}_1 = e^S{}_1, ..., \mathrm{v}_n = e^S{}_n$

Function g^S: $U^S \times S^S \rightarrow S^S$ applies updates of type u^S to some state s^S as follows:

$$g^S(v_1 = e^S{}_1, ..., v_n = e^S{}_n, s^S)(v) = \begin{cases} s^S(v) & \text{if v} \notin \{v_1, ..., v_n\} \\ f^S(e^S{}_i, s^S) & \text{if v} = v_i \text{ and} \\ & v_i \in \{v_1, v_2, ..., v_n\} \end{cases}$$

5.3 RELAX

RELAX is an instance of *System* of the form $\langle T^R, ex^R, f^R, g^R \rangle$. T^R is a transition system that represents the circuit implementation after rule numbering and relaxation.

After relaxation, the enabling condition of each rule R tests the state of the system before the earliest rule R' executes, whose execution does not disable the original enabling condition of R.

We define a new **external** function ex^R that has the same values as $ex^S(t_i)$ for each transition $t_i \in T^R$ with the same label as the transition in T. We defer the definition of the external function for the newly introduced transitions to the construction of the new transition system.

A **state** s is a function $s : (Vars \times Versions) \cup \{\mathsf{pc}\} \rightarrow Vals$.

pc is a variable of type integer that represents the ordinal number of the transition in the relaxed implementation that is currently under evaluation. pc provides a way to express the deterministic execution of the transitions in the relaxed circuit, following the order used by the relaxation transformation.

Versions is a set of integers.

The **set** S^R **of states** $s^R \in S^R$ is the set of functions $S^R = (Vars \times Versions) \cup \{\mathsf{pc}\} \rightarrow Vals$.

Our transition system contains positions, while the state does not. A position and variable name pair uniquely denotes a state variable instance

within the condition of a transition.

We redefine the set of expressions E^S in SPEC to be E^R in RELAX

$e^R ::= c \mid (v, n, p) \mid \mathsf{pc} \mid \rho(e^R{}_1, ..., e^R{}_n)$

The triple (v, n, p) stands for a variable $v \in Vars$, its version $n \in Versions$ and position $p \in Position$. $Position$ is a set of integers.

Function $f^R : E^R \times S^R \rightarrow Vals$ evaluates expressions of type e^R in some state s^R as follows:

$f^R(c, s^R) = c$

$f^R((v, n, p), s^R) = s^R(v, n)$

$f^R(\mathsf{pc}, s^R) = s^R(\mathsf{pc})$

$f^R(\rho(e^R{}_1, ..., e^R{}_n), s^R) = \rho(f^R(e^R{}_1, s^R), ..., f^R(e^R{}_n, s^R))$

We also redefine the set of updates U^S in SPEC to U^R in RELAX

$ur ::= \mathsf{pc}{+}{+}$

$\mid \mathsf{pc} = 1$

$\mid (v_1, nr_1, p_1) = e^R{}_1, ..., (v_n, nr_n, p_n) = e^R{}_n, \mathsf{pc} = \mathsf{pc}{+}{+}$

$\mid (v_1, nr_1, p_1) = e^R{}_1, ..., (v_n, nr_n, p_n) = e^R{}_n, \mathsf{pc} = 1$

In our system, if a variable does not get updated by a transition, the designer does not have to specify that its value remains unchanged. We introduce a function **update**: $U^R \times S^R \rightarrow \mathcal{P}(Vars)$ which takes an update $u^R \in U^R$ and a state $s^R \in S^R$ and returns the set of variables in $Vars$ that get updated.

Function g^R: $U^R \times S^R \rightarrow S^R$ applies updates of type u^R to some state s^R as follows:

$$g^R((v_1, n_1, p_1) = e^R{}_1, ..., (v_m, n_m, p_m) = e^R{}_m, \mathsf{pc}{+}{+}, s^R)(\mathsf{pc}) = f^R(\mathsf{pc}{+}{+}, s^R)$$

$$g^R((v_1, n_1, p_1) = e^R{}_1, ..., (v_m, n_m, p_m) = e^R{}_m, \mathsf{pc} = 1, s^R)(\mathsf{pc}) = f^R(1, s^R)$$

$$g^R(\mathsf{pc}++, s^R)(\mathsf{pc}) = f^R(\mathsf{pc}++, s^R)$$

$$g^R(\mathsf{pc} = 1, s^R)(\mathsf{pc}) = f^R(1, s^R)$$

$$g^R(\mathsf{pc}++, s^R)((v, n)) = s^R(v, n)$$

$$g^R(\mathsf{pc} = 1, s^R)((v, n)) = s^R(v, n)$$

$$g^R((v_1, n_1, p_1) = e^R{}_1, ..., (v_m, n_m, p_m) = e^R{}_m, \mathsf{pc}++, s^R)(v, n) =$$

$$= \begin{cases} s^R((v, n)) & \text{if } v \notin \{v_1, ..., v_m\} \text{ or} \\ & n \notin \{n_1, ..., n_m\} \text{ or} \\ & s^R(\mathsf{pc}) \notin \{n_1 - 1, ..., n_m - 1\} \\ f^R(e^R{}_i, s^R) & \text{otherwise} \end{cases}$$

$$g^R((v_1, n_1, p_1) = e^R{}_1, ..., (v_m, n_m, p_m) = e^R{}_m, \mathsf{pc} = 1, s^R)(v, n) =$$

$$= \begin{cases} s^R((v, n)) & \text{if } v \notin \{v_1, ..., v_m\} \text{ or} \\ & n \notin \{n_1, ..., n_m\} \text{ or} \\ & s^R(\mathsf{pc}) \notin \{n_1 - 1, ..., n_m - 1\} \\ f^R(e^R{}_i, s^R) & \text{otherwise} \end{cases}$$

We define a numbering function RN: $(E^S \cup U^S) \times \text{Version} \to E^R \cup U^R$ as follows:

$RN(c)(n) = c$

$RN(v)(n) = (v, n, 0)$

$RN(\rho(e^S{}_1, ..., e^S{}_n))(n) = \rho(RN(e^S{}_1)(n), ..., RN(e^S{}_n)(n))$

$RN(\mathsf{pc}++)(n) = \mathsf{pc}++$

$RN(\mathsf{pc}=1)(n) = \mathsf{pc}=1$

$RN((v_1, nr_1, p_1) = e^R{}_1, ..., (v_n, nr_n, p_n) = e^R{}_n, \mathsf{pc}++)(k) =$

$((v_1, k+1, 0) = RN(e^S{}_1)(k); ...; (v_n, k+1, 0) = RN(e^S{}_n)(k), \mathsf{pc}++$

$RN((v_1, nr_1, p_1) = e^R{}_1, ..., (v_n, nr_n, p_n) = e^R{}_n, \mathsf{pc}=1)(k) =$

$((v_1, k+1, 0) = RN(e^S{}_1)(k); ...; (v_n, k+1, 0) = RN(e^S{}_n)(k), \mathsf{pc}=1$

We define a **variable positioning** function

$VP: E^R \times Position \rightarrow E^R \times Position$ as follows:

$VP(c, i) = (c, i)$

$VP((v, n, 0), i) = ((v, n, i), i+1)$

$VP(\mathsf{pc}, i) = (\mathsf{pc}, i)$

$VP(\rho(e^R{}_1, ..., e^R{}_n), i) =$

let

$(a_1, b_1) = VP(e^R{}_1, i)$

$(a_2, b_2) = VP(e^R{}_2, b_1)$

...

$(a_n, b_n) = VP(e^R{}_n, b_{n-1})$

in

$(\rho(a_1, a_2, ..., a_n), b_n)$

end

We can now define a relaxation function RE that replaces the current version of each variable $v \in Vars$ in position $p \in Position$ with its relaxed version. We obtain the relaxed version of a variable $v \in Vars$ with original version $n \in Versions$ in position $p \in Position$ by invoking a function $\sigma \in \Sigma : Vars \times Versions \times Position \rightarrow Versions$. The original version n is equal to $l \bmod |T|$, where l is the label of the transition that invokes $\sigma \in \Sigma$. From the construction of the new transition system we

will see that the tuple (v, n, p) is unique within the set of all the external transitions in T^R.

$RE : E^R \times \Sigma \rightarrow E^R$

$RE(c, \sigma) = c$

$RE((v, n, p), \sigma) = (v, \sigma(v, n, p), p)$

$RE(\mathsf{pc}, \sigma) = \mathsf{pc}$

$RE(\rho({e^R}_1, ..., {e^R}_n), \sigma) = \rho(RE({e^R}_1, \sigma), ..., RE({e^R}_n, \sigma))$

We define a new **transition system** $T^R = \{t_i \equiv l_i : c_i \Rightarrow u_i | i \in \{1, ..., n\}\}$ by modifying the previous transition system T as follows:

- For every transition $t \equiv l : c \Rightarrow u \in T$, construct two transitions in T^R, one external and one not external, as follows:
  $t_l \equiv l : (\mathsf{pc} == l) \wedge RE(\pi_1(VP(RN(c^S)(l), 1)), \sigma) \Rightarrow RN(u^S)(l); \mathsf{pc}++$
  $t_{|T|+l} \equiv |T| + l : (\mathsf{pc} == l) \wedge \overline{RE(\pi_1(VP(RN(c^S)(l), 1)), \sigma)} \Rightarrow \mathsf{pc}++$, where $|T| + l$ is a fresh label and $ex^R(t_{|T|+l}) = \mathsf{false}$
  We use π_1 for the projection of the first element of a tuple. For a double (a, b) we have $\pi_1(a, b) = a$.
- Create a new transition $t_{2*|T|+1}$ to express the wrap-around after the relaxation algorithm tried all other transitions and either executed them or not. The new transition is not external.
  $\forall v \in Vars\ t_{2*|T|+1} \equiv 2 * |T| + 1 : (\mathsf{pc} == |T| + 1) \Rightarrow s(v, 1) = s(v, s(\mathsf{pc}));\ \mathsf{pc} = 1$, where $2*|T|+1$ is a fresh label and $ex^R(t_{2*|T|+1}) = \mathsf{false}$

Assume there exists an **initial state** ${s^R}_0$ of T^R. In ${s^R}_0$ the following are true:

- $\forall v_k \in Vars - Queues, \forall i \in \{1, ..., |T| + 1\},\ {s^R}_0(v_k, i) = initialVal_k$, where $initialVal_k \in Vals$ are the initial values of the registers and memories in the circuit.
- ${s^R}_0(\mathsf{pc}) = 1$

- All the queues in the circuit specification are empty.
  $\forall k$ s.t. $1 \leq k \leq maxQueue,\ {s^R}_0(queue_k) = \{\}$

5.4 Correctness

To prove that the relaxation algorithms produces a correct result, we need to prove two properties: simulation and non-termination. That means to prove that the behavior of the resulting specification after relaxation simulates the specification before relaxation, and also that relaxation cannot stop the progress in the execution of the system.

5.4.1 Simulation

We want to prove that RELAX simulates SPEC. This means that we want to prove that for any execution in RELAX, we can find an execution in SPEC with the same execution sequence.

To do this we need to first define the abstraction function AF that maps each state of RELAX to a state of SPEC:

$$FUNC\ AF(s^R) \rightarrow \{\forall v \in Vars \mid s^S(v) = s^R(v, s^R(\mathsf{pc}))\}$$

Before formally stating the simulation theorem we prove some lemmas.

Observation: For each $\langle s^R, t^R, s^{R'} \rangle \in R^R$ where $t^R \equiv l^R : c \Rightarrow u$ and $ex^R(t^R) = \mathsf{true}$, we have $s^R(\mathsf{pc}) = l^R$.

Proof: True by construction of T^R.

Lemma 1: ConsistentExecutionTrace defined below is an invariant for the execution of T^R.

ConsistentExecutionTrace(s^R,T^R) =

$\mathsf{pc} > 1 \wedge f^R(RE(\pi_1(VP(RN({c^S}_1)(1), 1)), \sigma), s^R) \Rightarrow$

$$s^R(v,2) = g^R(RN(u^S{}_1)(1), s^R)(v,1);$$

$\wedge\ \mathsf{pc} > 1\ \wedge\ \overline{f^R(RE(\pi_1(VP(RN(c^S{}_1)(1),1)),\sigma), s^R)} \Rightarrow$

$$s^R(v,2) = s^R(v,1);$$

$\wedge\ \mathsf{pc} > 2 \wedge f^R(RE(\pi_1(VP(RN(c^S{}_2)(2),1)),\sigma), s^R) \Rightarrow$

$$s^R(v,3) = g^R(RN(u^S{}_2)(2), s^R(v,2));$$

$\wedge\ \mathsf{pc} > 2\ \wedge\ \overline{f^R(RE(\pi_1(VP(RN(c^S{}_2)(2),1)),\sigma), s^R)} \Rightarrow$

$$s^R(v,3) = s^R(v,2);$$

$\wedge\ ...$

$\wedge\ \mathsf{pc} > |T| \wedge f^R(RE(\pi_1(VP(RN(c^S{}_{|T|})(|T|),1)),\sigma), s^R) \Rightarrow$

$$s^R(v,|T|+1) = g^R(RN(u^S{}_{|T|})(|T|), s^R(v,|T|));$$

$\wedge\ \mathsf{pc} > |T|\ \wedge\ \overline{f^R(RE(\pi_1(VP(RN(c^S{}_{|T|})(|T|),1)),\sigma), s^R)} \Rightarrow$

$$s^R(v,|T|+1) = s^R(v,|T|);$$

$\wedge\ 1 \leq \mathsf{pc} \leq |T|+1$

Proof:

Basis: We know from the definition of $s^R{}_0$ that $s^R{}_0(\mathsf{pc}) = 1$. In this state ConsistentExecutionTrace(s^R,T^R) holds.

Induction Step:

Assume that ConsistentExecutionTrace(s^R,T^R) holds before some transition executes. This means $1 \leq \mathsf{pc} \leq |T|+1$. Let $s^R(pc) = k$. For this k we know that the following holds:

$\mathsf{pc} > 1 \wedge f^R(RE(\pi_1(VP(RN(c^S{}_1)(1),1)),\sigma), s^R) \Rightarrow$

$$s^R(v,2) = g^R(RN(u^S{}_1)(1), s^R)(v,1);$$

$\wedge\ \mathsf{pc} > 1\ \wedge\ \overline{f^R(RE(\pi_1(VP(RN(c^S{}_1)(1),1)),\sigma), s^R)} \Rightarrow$

$$s^R(v,2) = s^R(v,1);$$

$\wedge\ \mathsf{pc} > 2 \wedge f^R(RE(\pi_1(VP(RN(c^S{}_2)(2),1)),\sigma), s^R) \Rightarrow$

$$s^R(v,3) = g^R(RN(u^S{}_2)(2), s^R(v,2));$$

$$\wedge\ \mathsf{pc} > 2\ \wedge\ \overline{f^R(RE(\pi_1(VP(RN(c^S{}_2)(2),1)),\sigma), s^R)} \Rightarrow$$

$$s^R(v,3) = s^R(v,2);$$

$$\wedge\ ...$$

$$\wedge\ \mathsf{pc} > k-1 \wedge f^R(RE(\pi_1(VP(RN(c^S{}_{k-1})(k-1),1)),\sigma), s^R) \Rightarrow$$

$$s^R(v,k) = g^R(RN(u^S{}_{k-1})(k-1), s^R(v,k-1));$$

$$\wedge\ \mathsf{pc} > k-1\ \wedge\ \overline{f^R(RE(\pi_1(VP(RN(c^S{}_{k-1})(k-1),1)),\sigma), s^R)} \Rightarrow$$

$$s^R(v,k) = s^R(v,k-1);$$

We want to prove that ConsistentExecutionTrace(s^R,T^R) holds after the next transition. We do a case analysis on the value of k.

- $k = |T| + 1$

  In this case $t^R{}_{2*|T|+1}$ of the form

  $$t_{2*|T|+1} \equiv 2 * |T| + 1 : (s^R(\mathsf{pc}) == |T| + 1) \Rightarrow\ s^R(v,1) = s^R(v, s^R(\mathsf{pc})), \quad s^R(\mathsf{pc}) = 1;$$

  executes and sets $s^R(\mathsf{pc})$ to 1. For this value of $s^R(\mathsf{pc})$ the invariant holds trivially.

- $k \neq |T| + 1$

  In this case either $t^R{}_k$ or $t^R{}_{|T|+k}$ will execute.

  - If $f^R(RE(\pi_1(VP(RN(c^S{}_k)(k),1)),\sigma)) = \mathsf{true}$ then $t^R{}_k$ executes and sets

    $$\forall v\ s^R(v,k+1) = g^R(RN(u^S)(k), s^R)(v,k)$$
    $$s^R(\mathsf{pc}) = s^R(\mathsf{pc}) + + \tag{5.1}$$

    It directly results that all values of $s^R(v,i)$ for $i \in \{1,...,k\}$ remain unchanged. Because $s^R(\mathsf{pc})$ becomes $k+1$, we have that the following are still true for any $i \in \{2,...,k\}$:

$$\mathsf{pc} > i-1 \ \wedge \ f^R(RE(\pi_1(VP(RN({c^S}_{i-1})(i-1),1)),\sigma),s^R) \Rightarrow$$
$$s^R(v,i) = g^R(RN({u^S}_{i-1})(i-1), s^R(v,i-1));$$
$$\mathsf{pc} > i-1 \ \wedge \ \overline{f^R(RE(\pi_1(VP(RN({c^S}_{i-1})(i-1),1)),\sigma),s^R)} \Rightarrow$$
$$s^R(v,i) = s^R(v,i-1); \tag{5.2}$$

From 5.5 we have that $s^R(\mathsf{pc})$ becomes $k+1$ and

$$\mathsf{pc} > k \ \wedge \ f^R(RE(\pi_1(VP(RN({c^S}_k)(k),1)),\sigma),s^R) \Rightarrow$$
$$s^R(v,k+1) = g^R(RN({u^S}_k)(k), s^R(v,k)); \tag{5.3}$$

is true. Also, because $k \neq |T|+1$ and $s^R(\mathsf{pc}) = k+1$ we have that

$$s^R(\mathsf{pc}) \leq |T|+1 \tag{5.4}$$

From 5.6, 5.7 and 5.8 we have that ConsistentExecutionTrace(s^R,T^R) holds after transition ${t^R}_k$.

- If $f^R(RE(\pi_1(VP(RN({c^S}_k)(k),1)),\sigma)) = \mathsf{false}$ then ${t^R}_{|T|+k}$ executes and sets

  vspace-0.2in

  $$s^R(\mathsf{pc}) = s^R(\mathsf{pc}) + +$$ (5.5)

  It directly results that all values of $s^R(v,i)$ for $i \in \{1,...,k\}$ remain unchanged. Because $s^R(\mathsf{pc})$ becomes $k+1$, we have that the following are still true for any $i \in \{2,...,k\}$:

$$\mathsf{pc} > i-1 \ \wedge \ f^R(RE(\pi_1(VP(RN({c^S}_{i-1})(i-1),1)),\sigma),s^R) \Rightarrow$$
$$s^R(v,i) = g^R(RN({u^S}_{i-1})(i-1), s^R(v,i-1));$$
$$\mathsf{pc} > i-1 \ \wedge \ \overline{f^R(RE(\pi_1(VP(RN({c^S}_{i-1})(i-1),1)),\sigma),s^R)} \Rightarrow$$
$$s^R(v,i) = s^R(v,i-1); \tag{5.6}$$

From 5.5 we have that $s^R(\mathsf{pc})$ becomes $k+1$ and

$$\mathsf{pc} > k \ \wedge \ \overline{f^R(RE(\pi_1(VP(RN(c^S{}_k)(k),1)),\sigma),s^R)} \Rightarrow$$
$$s^R(v,k+1) = s^R(v,k) \tag{5.7}$$

is true. Also, because $k \neq |T|+1$ and $s^R(\mathsf{pc}) = k+1$ we have that

$$s^R(\mathsf{pc}) \leq |T|+1 \tag{5.8}$$

From 5.6, 5.7 and 5.8 we have that ConsistentExecutionTrace(s^R,T^R) holds after transition $t^R{}_{|T|+k}$.

This concludes the proof of Lemma 1.

Lemma 2:

$\forall s^R \in S^R,\ \forall s^S \ \in S^S,\ \forall t^R,\ \forall s^{R'} \ \in S^R,\ \forall e^S \in E^S \ .\ s^S \ = AF(s^R),$
if $\langle s^R, t^R, s^{R'} \rangle \ \in \ R^R$ then
$$f^S(e^S,s^S) = f^R(RN(e^S)(i),s^R), \quad \text{where } i = s^R(\mathsf{pc})$$

Proof: We prove the lemma by structural induction on e^S.

- $e^S = c$
  From the definition of RN and f^R: $f^R(RN(e^S)(i),s^R) = f^R(c,s^R) = c$
  From the definition of f^S: $f^S(c,s^S) = c$
  Therefore $f^S(e^S,s^S) = f^R(RN(e^S)(i),s^R)$
- $e^S = v$
  From the definition of RN: $f^R(RN(e^S)(i),s^R) \ = f^R(RN(v)(i),s^R) \ = f^R((v,i),s^R)$
  From the definition of f^R: $f^R((v,i),s^R) \ = \ s^R(v,i)$
  From the definition of f^S: $f^S(e^S,s^S) = f^S(v,s^S) = s^S(v)$
  But from the definition of AF: $\forall v \ s^S(v) = s^R(v,i)$
  Therefore $f^S(e^S,s^S) = f^R(RN(e^S)(i),s^R)$
- $e^S = \rho(e^S{}_1, ..., e^S{}_n)$

$f^R(RN(e^S)(i),s^R) = f^R(\rho(RN(e^S{}_1)(\mathrm{i}),...,RN(e^S{}_n)(\mathrm{i})),s^R)$

From the definition of f^R:

$f^R(\rho(RN(e^S{}_1)(\mathrm{i}),...,RN(e^S{}_n)(\mathrm{i})),s^R) =$

$$\rho(f^R(RN(e^S{}_1)(\mathrm{i}),s^R),...,f^R(RN(e^S{}_n)(\mathrm{i}),s^R))$$

From the definition of f^S:

$$f^S(\rho(e^S{}_1,...,e^S{}_n),s^S) = \rho(f^S(e^S{}_1,s^S),...,f^S(e^S{}_n,s^S))$$

But we know that $\forall i \in \{1,...,n\}\ f^R(RN(e^S{}_i)(\mathrm{i}),s^R) = f^S(e^S{}_i,s^S)$. This is true because $e^S{}_i$ is either $c \mid v$, for which we already proved the lemma, or a building block of the same type $(\rho'((e^{S'}{}_1,...,e^{S'}{}_n))$.

Therefore $f^S(e^S,s^S) = f^R(RN(e^S)(i),s^R)$

This concludes the proof for Lemma 2.

Simulation Theorem:

We formally define the simulation theorem as follows:

$$\forall s^R \in S^R,\ \forall s^S\ \in S^S,\ \forall t^R,\ \forall s^{R'} \in S^R\ .\ s^S\ = AF(s^R),$$
$$\text{if } \langle s^R, t^R, s^{R'}\rangle\ \in\ R^R \text{ then}$$
$$(\exists t^S,\ \exists s^{S'}\ \in S^S\ .\ \langle s^S, t^S, s^{S'}\rangle\ \in\ R^S \text{ and } s^{S'} = AF(s^{R'}))$$
$$\text{or } s^S = AF(s^{R'})$$

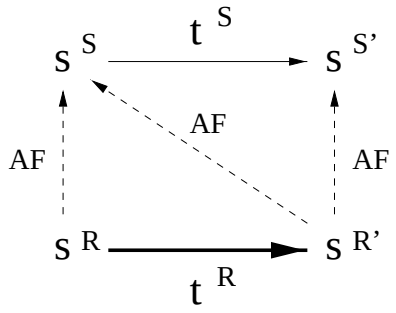

Figure 5.1: Commutative Diagram for Simulation

Simulation Proof:

We develop a proof by induction on the length of the execution sequence.

1. Basis:

$AF(s^R_0) = s^S_0$

This is trivially true by the definition of s^R_0 and s^S_0.

For each $v_k \in Vars$,

$$s^R{}_0(v_k, s^R{}_0(\mathsf{pc})) = initialVal_k = s^S{}_0(v_k)$$

By definition of AF:

$$AF(s^R{}_0) = s^R{}_0(v_k, s^R{}_0(\mathsf{pc})), \forall v_k \in Vars$$

Therefore, $AF(s^R{}_0) = s^S{}_0$.

2. Induction step:

We do a case analysis on $ex^R(t^R)$:

- $ex^R(t^R) = \mathsf{false}$, where $t^R \equiv l^R : c^R \Rightarrow u^R$

  From the hypothesis

  $$\langle s^R, t^R, s^{R'} \rangle \ \in R^R \ . \ ex^R(t^R) = \mathsf{false}$$

  From this and the definition of T^R, either:

  - $t^R \equiv l^R : (\mathsf{pc} == l^R) \wedge \overline{RE(\pi_1(VP(RN(c^S)(l^R), 1)), \sigma)} \Rightarrow \mathsf{pc}++$

    From the semantics of t^R:

    $$g^R(\mathsf{pc}++, s^R) = s^{R'} \tag{5.9}$$

    From the definition of AF:

    $$\forall v \ AF(s^{R'})(v) = s^{R'}(v, s^{R'}(\mathsf{pc}))$$

    Substituting 5.9 in the equality above, we obtain:

    $$AF(s^{R'})(v) = g^R(\mathsf{pc}++, s^R)(v, g^R(\mathsf{pc}++, s^R)(\mathsf{pc}))$$

    From the definition of g^R, the equality becomes:

    $$AF(s^{R'})(v) = g^R(\mathsf{pc}++, s^R)(v, f^R(\mathsf{pc}++, s^R))$$

We apply the definition of g^R again and obtain:

$$AF(s^{R'})(v) = s^R(v, f^R(\mathsf{pc}++, s^R)) \tag{5.10}$$

From the definition of f^R:

$$f^R(\mathsf{pc}++, s^R) = f^R(\mathsf{pc}, s^R) + 1 = s^R(\mathsf{pc})++$$

From this and 5.10 we have

$$AF(s^{R'})(v) = s^R(v, s^R(\mathsf{pc})++)$$

But t^R only modifies pc and therefore

$$s^R(v, s^R(\mathsf{pc})++) = s^R(v, s^R(\mathsf{pc}))$$

Therefore

$$AF(s^{R'})(v) = s^R(v, s^R(\mathsf{pc})) \tag{5.11}$$

We know that s^S= AF(s^R). From the definition of AF: $\forall v\ s^S$(v) = s^R(v,s^R(pc)). From this and 5.11 we have

$$\forall v\ AF(s^{R'})(v) = s^S(v)$$

- $t^R \equiv l^R : (\mathsf{pc} == 2 * |T| + 1) \Rightarrow\ s(v, 1) = s(v, \mathsf{pc});\ \mathsf{pc} = 1$

  From the semantics of t^R:

$$g^R((v, 1, p) = (v, s^R(\mathsf{pc})); \mathsf{pc} = 1, s^R) = s^{R'} \tag{5.12}$$

From the definition of AF:

$$\forall v\ AF(s^{R'})(v) = s^{R'}(v, s^{R'}(\mathsf{pc}))$$

Substituting 5.12 in the equality above we obtain:

$$AF(s^{R'})(v) = g^R((v, 1, p) = (v, s^R(\mathsf{pc})); \mathsf{pc} = 1, s^R)$$

$$(v, g^R((v, 1, p) = (v, s^R(\mathsf{pc})); \mathsf{pc} = 1, s^R)(\mathsf{pc})) \tag{5.13}$$

From the definition of g^R:

$$g^R((v, 1, p) = (v, s^R(\mathsf{pc})); \mathsf{pc} = 1, s^R)(\mathsf{pc}) = f^R(1, s^R)$$

which from the definition of f^R is 1. Therefore 5.13 becomes

$$AF(s^{R'})(v) = g^R((v, 1, p) = (v, s^R(\mathsf{pc})); \mathsf{pc} = 1, s^R)(v, 1)$$

From the definition of g^R:

$$AF(s^{R'})(v) = f^R((v, s^R(\mathsf{pc})), s^R)$$

which from the definition of f^Rbecomes

$$AF(s^{R'})(v) = s^R(v, s^R(\mathsf{pc})) \tag{5.14}$$

We know that s^S= AF(s^R). From the definition of AF: $\forall v\ s^S$(v) = s^R(v,s^R(pc)). From this and 5.14 we have

$$\forall v\ AF(s^{R'})(v) = s^S(v)$$

- $ex^R(t^R) = \mathsf{true}$, where $t^R \equiv l^R : c^R \Rightarrow u^R$

  From the hypothesis

  $$\langle s^R, t^R, s^{R'} \rangle \ \in \ R^R \text{ and } ex^R(t^R) = \mathsf{true}$$

  This implies:

  $$t^R \equiv l^R : (\mathsf{pc} == l^R) \ \wedge \ RE(\pi_1(VP(RN(c^S)(l^R), 1)), \sigma) \Rightarrow$$
  $$RN(u^S)(l^R); \mathsf{pc}++$$

  By the semantics of t^R the following are true:

  $$f^R((\mathsf{pc} == l^R) \ \wedge \ RE(\pi_1(VP(RN(c^S)(l^R), 1)), \sigma), s^R)$$
  $$g^R(RN(u^S)(l^R); \mathsf{pc}++ \ , \ s^R) = s^{R'}$$

  These imply:

  $$f^R(RE(\pi_1(VP(RN(c^S)(l^R), 1)), \sigma), s^R) \text{ and}$$
  $$f^R(\mathsf{pc} == l^R, s^R) \text{ and}$$
  $$g^R(RN(u^S)(l^R); \mathsf{pc}++ \ , \ s^R) = s^{R'} \tag{5.15}$$

  The relaxation technique preserves the following property:

$$\forall s^R \in S^R,\ \forall i\ .\ \text{ConsistentExecutionTrace}(s^R, T^R) \text{ and}$$
$$f^R(RE(\pi_1(VP(RN(c^S{}_i)(l_i), 1)), \sigma), s^R)\ \wedge\ (\mathsf{pc} \geq l_i)$$
$$\Longrightarrow f^R(\pi_1(VP(RN(c^S)(l_i), 1)), s^R) \tag{5.16}$$

From 5.15 and 5.16 we infer

$$\forall s^R \in S^R, \forall i.\ f^R(\pi_1(VP(RN(c^S)(l_i), 1)), s^R) \text{ and}$$
$$g^R(RN(u^S{}_i)(l_i); \mathsf{pc}++,\ s^R) = s^{R'}$$

What we want to prove is that:

$$\exists s^{S'} \in S^S,\ \exists j\ .\ s^{S'} = AF(s^{R'}) \text{ and } t_j \equiv l_j : c_j \Rightarrow u_j \in T^S \text{ and}$$
$$f^S(c_j, s^S) \text{ and } g^S(u_j, s^S) = s^{S'}$$

We choose $j = i$ and $s^{S'} = g^S(u_i, s^S)$.

From **Lemma 2** we have that $f^S(c^S{}_i, s^S) = f^R(RN(c^S{}_i)(i), s^R)$.

We know that $f^R(RN(c^S{}_i)(i), s^R) = \mathsf{true}$ and therefore $f^S(c^S{}_i, s^S) = \mathsf{true}$.

We still have to prove that $s^{S'} = \text{AF}(s^{R'})$.

We know the following are true:

(1) $s^S = \text{AF}(s^R)$

(2) $s^{R'} = g^R(RN(u^S{}_i)(\text{i}); \mathsf{pc}{+}{+}, s^R)$

(3) $s^{S'} = g^S(u^S{}_i, s^S)$

We want to prove that $\forall\ s^R \forall\ s^S\ .\ s^S = \text{AF}(s^R) \Rightarrow \exists\ s^{R'}\ .\ s^{S'} = \text{AF}(s^{R'})$

From the definition of AF, this is equivalent to proving that

$$\forall v\ .\ s^{S'}(v) = s^{R'}(v, s^{R'}(\mathsf{pc}))$$

From (2), it is equivalent to

$$\forall v\ .\ s^{S'}(v) = g^R(RN(u^S)(i); \mathsf{pc}{+}{+}, s^R)(v, g^R(RN(u^S)(i); \mathsf{pc}{+}{+}, s^R)(\mathsf{pc}))$$

But from the definition of g^R we know that

$$g^R(RN(u^S{}_i)(i); \mathsf{pc}++, s^R)(\mathsf{pc}) = s^R(\mathsf{pc}) + 1$$

Therefore, we want to prove that

$$\forall v \ . \ s^{S'}(v) = g^R(RN(u^S)(i); \mathsf{pc}++, s^R)(v, s^R(\mathsf{pc}) + 1)$$

From (3), this is equivalent to

$$\forall v \ . \ g^S({u^S}_i, s^S)(v) = g^R(RN({u^S}_i)(i); \mathsf{pc}++, s^R)(v, s^R(\mathsf{pc}) + 1)$$

- $v \notin update(s^R, RN({u^S}_i)(i))$

  In other words, v does not get modified by the transition with label i. In this case, because $s^R(\mathsf{pc}) = i$ then $s^R(v, s^R(\mathsf{pc}) + 1) = s^R(v, s^R(\mathsf{pc}))$.

  From (1) and the definition of AF, the equality above becomes

  $$s^S(v) = s^R(v, s^R(\mathsf{pc}) + 1) \tag{5.17}$$

  From the definition of g^S: $g^S({u^S}_i,s^S)(\mathrm{v}) = s^S(\mathrm{v})$

  From the definition of $g^R$and because $v \notin update(s^R, RN({u^S}_i)(i))$ and $s^R(\mathsf{pc}) + 1 \neq i$:

  $$g^R(RN({u^S}_i)(i); \mathsf{pc}++, s^R)(v, s^R(\mathsf{pc}) + 1) = s^R(v, s^R(\mathsf{pc}) + 1)$$

  But 5.17 says that $\forall$ v $s^S(\mathrm{v}) = s^R(\mathrm{v}, s^R(\mathsf{pc})+1)$. Therefore

  $$\forall v \ . \ g^S({u^S}_i, s^S)(v) = g^R(RN({u^S}_i)(i); \mathsf{pc}++, s^R)(v, s^R(\mathsf{pc}) + 1)$$

- If $v \in update(s^R, RN({u^S}_i)(i))$

  In other words, v does get modified by the transition with label i.

  a. ${u^S}_i ::= \mathrm{v} = e^S$

  We want to prove that

  $\forall v \ g^S({u^S}_i, s^S)(v) = g^R((v, i+1) = RN(e^S)(i); \mathsf{pc}{+}{+}, s^R)(v, s^R(\mathsf{pc})+1)$

  From the definition of g^S: $g^S({u^S}_i,s^S)(\mathrm{v}) = f^S(e^S,s^S)$

  From **Observation** we know that

  $$s^R(\mathsf{pc}) = i \tag{5.18}$$

  From $v \in update(s^R, RN({u^S}_i)(i))$ and 5.18 we infer that we can use the `otherwise` branch of the definition of g^R and what we have to prove becomes

$$f^S(e^S, s^S) = f^R(RN(e^S)(i), s^R)$$

This is immediately true by **Lemma 2**.

b. ${u^S}_i$ ::= v_1=${e^S}_1$,...,v_n=${e^S}_n$

We claim that

$$v \in \text{ update}(s^R, RN({u^S}_i)(i)) \text{ iff } \exists!\ u \equiv v = e^S \in {u^S}_i \quad (5.19)$$

1. $$v \in \text{ update}(s^R, RN({u^S}_i)(i)) \Rightarrow \exists!\ u \equiv v = e^S \in {u^S}_i \quad (5.20)$$

The proof is by induction on ${u^S}_i$:

- Base case: ${u^S}_i = \mathrm{v}'$=e^S

  In this case, because $v \in update(s^R, RN({u^S}_i)(i))$, then v = v$'$ and therefore u=${u^S}_i$.

- Induction step: Assume that 5.20 holds for all u . $|u| < |{u^S}_i|$. If $v \in update(s^R, RN({u^S}_i)(i))$, then $\exists k$. $v \in update(s^R, RN({u^S}_i)(i))$. This means

  $$\exists k\ .\ u_k \equiv v = {e^S}_k \quad (5.21)$$

  We also know that all v_ks in ${u^S}_i$ are distinct. From this and 5.21 we infer that $\exists! u \equiv v$ =${e^S}$∈${u^S}_i$.

2. $\exists!\ u \equiv v = e^S \in {u^S}_i \Rightarrow v \in update(s^R, RN({u^S}_i)(i))$

True by definition of *update*.

Let $u_k \equiv v$ =${e^S}_k$ ∈${u^S}_i$ be the update u in 5.19.

We directly apply the result in case (a) above and obtain:

$$\forall v\ .\ g^S({u^S}_k, s^S)(v) = g^R(RN({u^S}_k)(k); \mathsf{pc}++, s^R)(v, s^R(\mathsf{pc}) + 1) \quad (5.22)$$

From the definition of g^S:

$$g^S({u^S}_k, s^S)(v) = f^S({e^S}_k, s^S) \quad (5.23)$$

and also

$$g^S({u^S}_i, s^S)(v) = f^S({e^S}_k, s^S) \quad (5.24)$$

From the definition of g^R:

$$g^R(RN({u^S}_k)(k); \mathsf{pc}++, s^R)(v, s^R(\mathsf{pc})+1) = f^R({e^S}_k, s^R) \text{ and also} \tag{5.25}$$

$$g^R(RN({u^S}_k)(k); \mathsf{pc}++, s^R)(v, s^R(\mathsf{pc})+1) = f^R({e^S}_k, s^R) \tag{5.26}$$

From 5.22, 5.23, 5.24, 5.25 and 5.26 we infer that

$$\forall v \ . \ g^S({u^S}_i, s^S)(v) = g^R(RN({u^S}_i)(i); \mathsf{pc}++, s^R)(v, s^R(\mathsf{pc})+1)$$

5.4.2 Non-termination

We want to prove that RELAX preserves the non-termination property of SPEC. Non-termination says that if from any state s of SPEC there exists an execution step that can be taken, then there exists an execution step in RELAX that can be taken from some state in RELAX reachable only by internal transitions from any state that maps to s using AF.

Let

$$R_0 = \{\langle s_1,\ s_2\rangle | \langle s_1,\ t,\ s_2\rangle \ \in R^R \text{ and } ex^R(t) = \mathsf{false}\}$$

We want to prove that:
$(\forall s^S \in S^S, \forall s^R \in S^R)$.

(ConsistentExecutionTrace(s^R, T^R) and $AF(s^R) = s^S$

if $\exists t^S \ . \ \langle s^S,\ t^S,\ {s^S}'\rangle \ \in R^S \Rightarrow$

$\exists t^R \ \exists s^{R*}$.

$\langle s^R,\ s^{R*}\rangle \ \in R_0^{3*|T|+1}$ and $\langle s^{R*},\ t^R,\ {s^R}'\rangle \ \in R^R$ and $ex^R(t^R) = \mathsf{true}$)

We first prove the following lemma:

Lemma 3: For any $k \geq 0$ and some state $s^R \in \ S^R$ s.t. $AF(s^R) \ = s^S$ and $l \ = s^R(\mathsf{pc})$, there exists a sequence of states s^{R0},...,s^{Rk} in RELAX starting from $s^{R0} = s^R$ s.t. the following hold:

$$\langle {s^R}^{k-1},\ {s^R}^k\rangle \ \in R_0^1 \tag{5.27}$$

$$\langle s^R,\ {s^R}^k\rangle \ \in \ R_0^k \tag{5.28}$$

$$s^S = AF(s^{R^k}) \tag{5.29}$$

$$\forall v \; s^{R^k}(v, l + d(k,l)) = s^R(v,l) \tag{5.30}$$

where k is the number of transition executions in RELAX

$$d(k,l) = \begin{cases} k & \text{if } k \leq |T| + 1 - l \\ k - (|T| + 1) & \text{otherwise} \end{cases}$$

Proof:

We present a proof by induction on the number of transitions executions in RELAX.

Induction Basis: $k = 0$

- $\langle s^R, \, {s^R}_0 \rangle \; = \; \langle {s^R}_0, \, {s^R}_0 \rangle \; \in \; R_0^0$
- From the definition of AF we have $s^S = AF(s^R)$. But $s^{R0} = s^R$ from hypothesis. Therefore $s^S = AF({s^R}_0)$
- $s^{R0} = s^R$ from hypothesis and therefore $\forall v \; {s^R}_0(v,0) \; = s^R(v,0)$

Induction Step:

Assume we know 5.27, 5.28, 5.29 and 5.30 hold for $k-1$:

$$\langle s^{R^{k-2}}, \; s^{R^{k-1}} \rangle \; \in R_0^1 \tag{5.31}$$

$$\langle s^R, \; s^{R^{k-1}} \rangle \; \in \; R_0^{k-1} \tag{5.32}$$

$$s^S = AF(s^{R^{k-1}}) \tag{5.33}$$

$$\forall v \; s^{R^{k-1}}(v, l + d(k,l) - 1) = s^R(v,l) \tag{5.34}$$

We show that they also hold for k:

$$\langle s^{R^{k-1}}, \; s^{R^k} \rangle \; \in R_0^1 \tag{5.35}$$

$$\langle s^R, \; s^{R^k} \rangle \; \in \; R_0^k \tag{5.36}$$

$$s^S = AF(s^{R^k}) \tag{5.37}$$

$$(\forall v \; s^{R^k}(v, l + d(k,l)) = s^R(v,l) \tag{5.38}$$

We know that $s^{Rk-1}(\mathsf{pc}) = l + d(k,l) - 1$. There are two cases:

- $s^{Rk-1}(\mathsf{pc}) \neq |T| + 1$.

  We assumed no external transition executes, and therefore the transition

$$
\begin{aligned}
t^R{}_{l+d(k,l)-1} \equiv (l + d(k,l) - 1) : \\
RE(\pi_1(VP(RN(c^S{}_{l+d(k,l)-1})(l + d(k,l) - 1), 1)), \sigma) \ \wedge \\
(s^{Rk-1}(\mathsf{pc}) = (l + d(k,l) - 1)) \Rightarrow \\
RN(u^S{}_{l+d(k,l)-1})(l + d(k,l) - 1); \mathsf{pc}{+}{+}
\end{aligned}
$$

  does not execute in s^{Rk-1}.

  From the semantics of a transition, this means that

  $f^R(RE(\pi_1(VP(RN(c^S{}_{l+d(k,l)-1})(l + d(k,l) - 1), 1)), \sigma), s^{Rk-1}) = \mathsf{false}$

  Therefore

  $f^R(\overline{RE(\pi_1(VP(RN(c^S{}_{l+d(k,l)-1})(l + d(k,l) - 1), 1)), \sigma)}, s^{Rk-1}) = \mathsf{true}$

  This means that the internal transition $t^R{}_{|T|+l+d(k,l)-1}$ executes and goes into a state s^{Rk}. This implies 5.35 is true:

$$\langle s^{Rk-1},\ s^{Rk} \rangle \ \in R_0^1$$

  The execution of $t^R{}_{|T|+l+d(k,l)-1}$ performs the following updates:

  $s^{Rk}(\mathsf{pc}) = s^{Rk-1}(\mathsf{pc}) + + = l + d(k,l)$

  $\forall v \ s^{Rk}(v, l + d(k,l)) = s^{Rk-1}(v, l + d(k,l) - 1)$

  By induction hypothesis we infer 5.38:

$$\forall v \ s^{Rk}(v, l + d(k,l)) = s^R(v, l)$$

  We know that $\langle s^{Rk-1},\ s^{Rk} \rangle \ \in R_0^1$. But $\langle s^R,\ s^{Rk-1} \rangle \ \in R_0^{k-1}$ and therefore 5.36 is true:

$$\langle s^R,\ s^{Rk} \rangle \ \in R_0^k$$

  From the hypothesis we have that $s^S = AF(s^{Rk-1})$ and therefore

$$s^S = s^{Rk-1}(v, s^{Rk-1}(\mathsf{pc})) \tag{5.39}$$

But

$$s^{R^k}(v, s^{R^k}(\mathsf{pc})) = s^{R^k}(v, l + d(k,l)) = \\ s^{R^{k-1}}(v, l + d(k,l) - 1) = s^{R^{k-1}}(v, s^{R^{k-1}}(\mathsf{pc})) \quad (5.40)$$

From 5.39 and 5.40 we have that 5.37 is true:

$$s^S = s^{R^k}(v, s^{R^k}(\mathsf{pc})) = AF(s^{R^k})$$

- $s^{Rk-1}(\mathsf{pc}) = |T| + 1$

  In this case $t^R{}_{2*|T|+1}$ executes and goes into a state s^{Rk}. This implies 5.35 is true:

$$\langle s^{R^{k-1}},\ s^{R^k} \rangle\ \in R_0^1$$

  The execution of $t^R{}_{2*|T|+1}$ performs the following updates: $s^{Rk}(\mathsf{pc}) = 1$ and $\forall v\ s^{Rk}(v,1) = s^{Rk-1}(v, l+d(k,l)-1)$. By induction hypothesis we infer 5.38:

$$\forall v\ s^{R^k}(v, l + d(k,l)) = s^R(v,l)$$

  We know that $\langle s^{Rk-1},\ s^{Rk} \rangle\ \in R_0^1$. But $\langle s^R,\ s^{Rk-1} \rangle\ \in R_0^{k-1}$ and therefore 5.36 is true:

$$\langle s^R,\ s^{R^k} \rangle\ \in R_0^k$$

  From the hypothesis we have that $s^S = AF(s^{Rk-1})$ and therefore

$$s^S\ =\ s^{R^{k-1}}(v, s^{R^{k-1}}(\mathsf{pc})) \quad (5.41)$$

  But

$$s^{R^k}(v, s^{R^k}(\mathsf{pc}))\ =\ s^{R^k}(v,1)\ = \\ s^{R^{k-1}}(v, l + d(k,l) - 1)\ =\ s^{R^{k-1}}(v, s^{R^{k-1}}(\mathsf{pc})) \quad (5.42)$$

  From 5.41 and 5.42 we have that 5.37 is true:

$$s^S = s^{R^k}(v, s^{R^k}(\mathsf{pc})) = AF(s^{R^k})$$

This completes the inductive proof of **Lemma 3**.

Non-termination Proof:

The proof proceeds by contradiction.

Choose $s^S \in S^S$ and:

$$s^R \in S^R \; . \; AF(s^R) = s^S \text{ and ConsistentExecutionTrace}(s^R, T^R)$$

If there exists at least one transition t^S s.t. $\langle s^S \; {t^S}_i \; s^{S'} \rangle \in R^S$, then there also exists such a transition with the smallest integer label.

Let $minI = min(\{l_i \mid \langle s^S, \; {t^S}_i, \; s^{S'} \rangle \in R^S, \; {t^S}_i \equiv l_i : c_i \Rightarrow u_i\})$ be the smallest integer label of all the transitions in SPEC that can execute from s^S.

Assume there exists no s^{R*} and t^R s.t. $\langle s^R, \; s^{R*} \rangle \in R_0^*$ and $\langle s^{R*}, \; t^R, \; s^{R'} \rangle \in R^R$ and t^R is an external transition.

Let k in Lemma 3 be $k = nrSteps = |T| + minI - l + 1$. Our assumption directly implies that there exists no s^{R*} and ${t^R}_i$ s.t. $\langle s^R, \; s^{R*} \rangle \in R_0^M$ for all $M \leq nrSteps$ where $\langle s^{R*}, \; {t^R}_i, \; s^{R'} \rangle \in R^R$ and t^R is an external transition.

If $l > minI$ then it takes $n = |T| - l + minI + 1$ steps to reach the point when the program counter is $minI$. One of these steps executes transition ${t^R}_{2*|T|+1}$, which wraps the values around. Because $n < nrSteps$, at step n we get to test the condition of transitions ${t^R}_i$ and ${t^R}_{i+|T|}$ and we know that

$$\forall v s^{R^n}(v, l + d(n, l)) = s^{R^n}(v, l + minI - 2) = ... = s^R(v, 1) \qquad (5.43)$$

If $l \leq minI$ then it takes $n = minI - l$ steps to reach the same point without performing the wrap-around. But we take $nrSteps$ transitions. From $l \leq minI$ we have that $nrSteps > |T|$. This means that one of

these steps executes transition ${t^R}_{2*|T|+1}$, which wraps the values around. Because $n < nrSteps$, at step n we get to test the condition of transitions ${t^R}_i$ and ${t^R}_{i+|T|}$ and we know that 5.43 holds again.

In $f^R(RE(\pi_1(VP(RN({c^S}_i)(i),1)),\sigma),s^{Rn})$, function σ returns the new version r for each pair of variable and position, where $1 \leq r \leq i$. From 5.43 we infer that $\forall v\ s^{Rn}(v,\sigma) = s^{Rn}(v,i)$.

This means that
$$f^R(RE(\pi_1(VP(RN({c^S}_i)(i),1)),\sigma),s^{R^n}) = f^R(\pi_1(VP(RN({c^S}_i)(i),1)),s^{R^n})$$

But applying VP does not affect the result of f^R and therefore
$$f^R(RE(\pi_1(VP(RN({c^S}_i)(i),1)),\sigma),s^{R^n}) = f^R(RN({c^S}_i)(i),s^{R^n}) \quad (5.44)$$

Because $s^{Rn}(\mathsf{pc}) = i$ and $s^S = AF(s^{Rn})$ (from 5.29), from **Lemma 2** we have that
$$f^S({c^S}_i,s^S) = f^R(RN({c^S}_i)(i),s^{R^n}) \quad (5.45)$$

From 5.44 and 5.45 we have that
$$f^R(RE(\pi_1(VP(RN({c^S}_i)(i),1)),\sigma),s^{R^n}) = f^S({c^S}_i,s^S)$$

But ${t^S}_i$ executes in SPEC and therefore $f^S({c^S}_i,s^S) = \mathsf{true}$

Therefore $f^R(RE(\pi_1(VP(RN({c^S}_i)(i),1)),\sigma),s^{Rn}) = \mathsf{true}$.

We also have $s^{Rn}(\mathsf{pc}) = i$. This means that the external transition ${t^R}_i$ executes from s^{Rn}, where $\langle s^R\ s^{Rn}\rangle \in R_0^n \in R_0^{nrSteps}$ (from 5.28).

But $nrSteps \leq 3 * |T| + 1$ and therefore $\langle s^R\ s^{Rn}\rangle \in R_0^n \in R_0^{3*|T|+1}$.

This contradicts our initial assumption and concludes the non-termination proof.

5.5 FINIT

Conceptually, the queues in our specification language have unbounded length. In hardware though, they must be implemented as finite hardware buffers. In our system, the designer specifies the desired length for

each of these hardware buffers. The challenge of the global scheduling algorithm is to enable the maximum number of rules for execution such that none of the queues in the system can overflow its designer-given length. The idea is that, during some clock cycle, a rule can insert into a full queue if the queue remains within its designer-specified length at the end of that clock cycle. More precisely, a rule can insert an element e into a queue, even if the queue is full, as long as enough following rules execute and remove items to make room for e in that queue.

The two inputs to the global scheduling transformation are a system $\langle T^R, ex^R, f^R, g^R \rangle$ of the type described in RELAX and a user-defined length $maxLength(queue_k)$ for each queue $queue_k, k \in \{1, ..., maxQueue\}$. The semantics of a transition is this system is the same as the semantics of a transition in RELAX with one difference:

If, for the given queue lengths, executing the update of the transition would overflow at least one of the queues, that transition is not enabled for execution, i.e:

$$[\![\langle c \Rightarrow u,\ T^R \rangle]\!] =$$
$$\{\langle s,\ c \Rightarrow u,\ g^R(u,s) \rangle\ .\ f^R(c,s) \ \text{ and } \ u \Rightarrow \text{overflow}(s_i) \text{ is } \mathsf{false}\},$$

where $\forall i \in \{s(\mathsf{pc}) + 1, ..., |T|\}$

$$s_i = \begin{cases} g^R(u_{i-1}, s_{i-1}) & \text{if } f^R(c_{i-1}, s_{i-1}) \\ s_{i-1} & \text{otherwise} \end{cases} \tag{5.46}$$

and overflow(s) is only defined for the states in which $s(pc) = |T| + 1$ as:

$$\text{overflow(s)} = \begin{cases} \exists queue_k \in Queues\ .\ length(queue_k, s) > maxLength(queue_k) \\ \qquad\qquad \text{if } s(\mathsf{pc}) = |T| + 1 \\ undefined \\ \qquad\qquad \text{otherwise} \end{cases}$$

and $length : Queues \times S^R \rightarrow Int$ is a function that returns the current length of the queue in the given state. We also define a function $room^R : Queues \times S^R \rightarrow Int$ that returns the number of empty slots of the given queue in the given state.

A transition system with this semantics defines a transition relation $R_q^R \subseteq S^R \times T^R \times S^R$. $R_q^R = \{\langle s^R, t^R, s^{R'} \rangle$. $t^R \equiv l_r : c^R \rightarrow u^R \in T^R \wedge f^R(c^R, s^R) \wedge g^R(u^R, s^R) = s^{R'} \wedge u^R \Rightarrow \text{overflow}(s^R{}_i)$ is $\mathsf{false}\}$, where $s^R{}_i$ is defined as in 5.46.

FINIT is an instance of $System$ of the form $\langle T^F, ex^F, f^F, g^F \rangle$. T^F is a transition system that represents the circuit implementation after global scheduling.

The set of expressions E^F in FINIT is:

$e^F ::= c \mid (v, n, p) \mid \mathsf{pc} \mid \rho(e^F{}_1, ..., e^F{}_n)$

The set of conditions C^F in FINIT is:

$c^F ::= \mathsf{true} \mid \mathsf{false} \mid (bv, n, p) \mid \varphi(e^F{}_1, ..., e^F{}_n)$

Function $f^F : (C^F \cup E^F) \times S^F \rightarrow (Bool \cup Vals)$ evaluates expressions of type e^F or c^F in some state s^F as follows:

$f^F(c \mid \mathsf{true} \mid \mathsf{false}, s^F) = c \mid \mathsf{true} \mid \mathsf{false}$

$f^F((v \mid bv, n, p), s^F) = s^F(v \mid bv, n)$

$f^F(\mathsf{pc}, s^F) = s^F(\mathsf{pc})$

$f^F(\rho(e^F{}_1, ..., e^F{}_n), s^F) = \rho(f^F(e^F{}_1, s^F), ..., f^F(e^F{}_n, s^F))$

$f^F(\varphi(e^F{}_1, ..., e^F{}_n), s^F) = \varphi(f^F(e^F{}_1, s^F), ..., f^F(e^F{}_n, s^F))$

The set of updates U^F in FINIT:

$u^F ::= \mathsf{pc}{+}{+}$

$\mid \mathsf{pc} = 1$

$\mid (v_1, nr_1, p_1) = e^F{}_1, ..., (v_n, nr_n, p_n) = e^F{}_n, \mathsf{pc}{+}{+}$

$\mid (v_1, nr_1, p_1) = e^F{}_1, ..., (v_n, nr_n, p_n) = e^F{}_n, \mathsf{pc} = 1$

The function $\mathbf{update^F} : U^F \times S^F \rightarrow \mathcal{P}(Vars)$ takes an update $u^F \in U^F$ and a state $s^F \in S^F$ and returns the set of variables in $Vars$ that get updated. Function $g^F : U^F \times S^F \rightharpoonup S^F$ applies updates of type u^F to some state s^F. Because of the semantics of transitions as defined above, g^F becomes a partial function and we define it as follows:

$$g^F((v_1, nr_1, p_1) = {e^F}_1, ..., (v_n, nr_n, p_n) = {e^F}_n, s^R)(\mathsf{pc}) = s^F(\mathsf{pc})$$

$$g^F(\mathsf{pc}{+}{+}, s^F)(\mathsf{pc}) = f^F(\mathsf{pc}{+}{+}, s^F)$$

$$g^F(\mathsf{pc} = 1, s^F)(\mathsf{pc}) = f^F(1, s^F)$$

$$g^F(\mathsf{pc}{+}{+}, s^F)((v, n)) = s^F(v, n)$$

$$g^F(\mathsf{pc} = 1, s^F)((v, n)) = s^F(v, n)$$

$$g^F((v_1, nr_1, p_1) = {e^F}_1, ..., (v_n, nr_n, p_n) = {e^F}_n, \mathsf{pc}{+}{+}, s^F)(v, n) =$$

$$= \begin{cases} s^F((v, n)) & \text{if } v \notin \{v_1, ..., v_n\} \text{ or} \\ & n \notin \{nr_1, ..., nr_n\} \text{ or} \\ & s^F(\mathsf{pc}) \notin \{nr_1 - 1, ..., nr_n - 1\} \\ f^F({e^F}_i, s^F) & \text{otherwise} \end{cases}$$

$$g^F((v_1, nr_1, p_1) = {e^F}_1, ..., (v_n, nr_n, p_n) = {e^F}_n, \mathsf{pc} = \mathsf{pc} + 1, s^F)(v, n) =$$

$$= \begin{cases} s^F((v, n)) & \text{if } v \notin \{v_1, ..., v_n\} \text{ or} \\ & n \notin \{nr_1, ..., nr_n\} \text{ or} \\ & s^R(\mathsf{pc}) \notin \{nr_1 - 1, ..., nr_n - 1\} \\ f^F({e^F}_i, s^R) & \text{otherwise} \end{cases}$$

We define two functions, $tail : t \times Queues \rightarrow Bool$ and $append : t \times Queues \rightarrow Bool$ that take a transition and a queue and return true iff the transition contains a *tail* or, correspondingly, an *append* operation on the queue given

as parameter. We also define a function $room^F : Queues \times S^F \rightarrow Int$ that returns the number of empty slots of the given queue in the given state.

We call a transition $t \equiv l : c \Rightarrow u \in T^F$ **appending** if

$$\exists queue_k \ \in \ Queues \ . \ append(t, queue_k) = \mathsf{true}$$

We define a new **transition system** $T^F = \{t_i \equiv l_i : c_i \Rightarrow u_i | i \in \{1, ..., n\}\}$ by modifying the previous transition system T^R such that for every appending transition $t \ \equiv l : \ c \ \Rightarrow \ u \ \in \ T^R$, we construct a transition in T^F of the form

$$t \ \equiv l : \ c \ \wedge \ finalLengthOK(s^F, T^F, \text{currentPath}) \ \Rightarrow \ u,$$

where currentPath starts as [].

Let *eval* be a function $eval : Bool \rightarrow \{0, 1\}$ which returns 1 for true and 0 for false.

For some appending transition $t^F \equiv l^F : c^F \Rightarrow u^F \ \in \ R^F$ from state s^F, we define:

$$finalLengthOKqueue(s^F, T^F, queue_k, \text{currentPath}) =$$
$$\sum_i eval(tail(t_i, queue_k) \bigwedge (f^F({c^F}_i \wedge finalLengthOK({s^F}_i, T^F, newPath), {s^F}_i)$$
$$\vee (t_i \in \text{currentPath}))) +$$
$$room(queue_k, s^F) > 0, \ \forall i \in \{l^F + 1, ..., |T|\} \ . \ tail(t_i, queue_k) = \mathsf{true},$$

where ${s^F}_{l^F} = s^F$ and

$${s^F}_i = \begin{cases} g^F({u^F}_{i-1}, {s^F}_{i-1}) & \text{if } f^F({c^F}'_{i-1}, {s^F}_{i-1}) \\ {s^F}_{i-1} & \text{otherwise} \end{cases}$$

$$newPath = \begin{cases} \text{currentPath} & \text{if } t_i \in \text{currentPath} \\ \text{currentPath} \vee t_i & \text{otherwise} \end{cases}$$

$$finalLengthOK(s^F, T^F, \text{currentPath}) =$$
$$\bigwedge finalLengthOKqueue(s^F, T^F, queue_k, \text{currentPath}),$$
$$\forall queue_k \in Queues \;.\; append(t^F, queue_k) = \mathsf{true}$$

currentPath holds the set of currently explored transitions for each starting transition in the system. This set is necessary for the case in which there exists cyclicity in the specification.

The semantics of a transition in $T^F$is the following:

If, for the given queue lengths, executing the update of the transition does not ensure that all the queues are within their maximum lengths when $s^F(\mathsf{pc}) = |T| + 1$, then T^Fgoes into an ERROR state s_{ERROR}.

We will show that the condition $finalLengthOK(s^F, T^F, \text{currentPath})$ makes sure that FINIT never goes into an ERROR state provided that the designer specifies appropriate lengths for all the queues.

Assume there exists an **initial state** s_0^F of T^F. In s_0^F the following are true:

- $\forall v_k \in Vars - Queues, \forall i \in \{1, ..., |T| + 1\},\ s_0^F(v_k, i) = initialVal_k$, where $initialVal_k \in Vals$ are the initial values of the registers and memories in the circuit.
- $s_0^F(\mathsf{pc}) = 1$
- All the queues in the circuit specification are empty.
  $\forall k$ s.t. $1 \leq k \leq maxQueue,\ s_0^F(queue_k) = \{\}$

5.5.1 Simulation

We want to prove that FINIT simulates RELAX, i.e. that for any execution in FINIT, we can find an execution in RELAX with the same execution sequence.

We first define an abstraction function AF^F that maps each state of FINIT to a state of RELAX. AF is the identity function:

$$FUNC\ AF^F(s^F)\ \rightarrow\ \{\forall v\ \in\ Vars \mid s^F(v,n)\ =\ s^R(v,n)\}$$

Lemma 4:

$\forall s^F \in S^F\ ,\ \forall s^R\ \in S^R\ ,\ \forall t^F\ ,\ \forall s^{F'} \in S^F\ .$
$s^R\ = AF(s^F)$ and $\langle s^F, t^F, s^{F'}\rangle\ \in\ R^F$ we have that
$f^F(e^F, s^F) = f^R(e^R, s^R)$, where $e^R \in E^R \cup C^R$ and $e^F \in E^F \cup C^F$

Proof: We prove the lemma by structural induction on e^R. Let $s^F(\mathsf{pc}) = \mathrm{i}$.

- $e^R = c \mid \mathsf{true} \mid \mathsf{false}$
  From the definition of f^F:
  $f^F(e^F, s^F) = f^F(c|\mathsf{true}|\mathsf{false}, s^F) = c|\mathsf{true}|\mathsf{false}$
  From the definition of f^R:
  $f^R(e^R, s^R) = f^R(c|\mathsf{true}|\mathsf{false}, s^R) = c|\mathsf{true}|\mathsf{false}$
  Therefore $f^R(e^R, s^R) = f^F(e^F, s^F)$
- $e^R = (v, n, p) \mid (bv, n, p)$
  From the definition of f^F:
  $f^F(e^F, s^F) = f^F((v|bv, n, p), s^F)\ =\ s^F(v|bv, n, p)$
  From the definition of f^R:
  $f^R(e^R, s^R)\ =\ f^R((v|bv, n, p), s^R) = s^R(v|bv, n, p)$
  But from the definition of AF: $\forall v\ s^R(v|bv, n, p) = s^F(v|bv, n, p)$
  Therefore $f^R(e^R, s^R) = f^F(e^F, s^F)$
- $e^R = \rho(e^R{}_1,...,e^R{}_n) \mid \varphi(e^R{}_1,...,e^R{}_n)$
  $f^F(e^F, s^F) = f^R(\rho(e^F{}_1, ..., e^F{}_n)|\varphi(e^F{}_1, ..., e^F{}_n, s^F)$
  From the definition of f^F:
  $f^F(\rho(e^F{}_1, ..., e^F{}_n), s^F) = \rho(f^F(e^F{}_1, s^F), ..., f^F(e^F{}_n, s^F))$ and
  $f^F(\varphi(e^F{}_1, ..., e^F{}_n), s^F) = \varphi(f^F(e^F{}_1, s^F), ..., f^F(e^F{}_n, s^F))$
  From the definition of f^R:
  $f^R(\rho(e^R{}_1, ..., e^R{}_n), s^R) = \rho(f^R(e^R{}_1, s^R), ..., f^R(e^R{}_n, s^R))$ and

$f^R(\varphi(e^R{}_1, ..., e^R{}_n), s^R) = \varphi(f^R(e^R{}_1, s^R), ..., f^R(e^R{}_n, s^R))$

But we know that $\forall i \in \{1, ..., n\}\ f^F(e^F{}_i, s^F) = f^R(e^R{}_i, s^R)$. This is true because $e^R{}_i$ is either $c\ |\mathsf{true}|\mathsf{false}\ |\ (v|bv, n, p)$, for which we already proved the lemma, or a building block of the same type $\rho'(e^{R'}{}_1, ..., e^{R'}{}_n)$ or $\varphi'(e^{R'}{}_1, ..., e^{R'}{}_n)$. Therefore $f^R(e^R, s^R) = f^F(e^F, s^F)$.

This concludes the proof for Lemma 4.

Simulation Theorem:

$$\forall s^F \in S^F,\ \forall s^R \in S^R,\ \forall t^F,\ \forall s^{F'} \in S^F\ .\ s^R\ = AF(s^F) \text{ and } \langle s^F, t^F, s^{F'}\rangle\ \in\ R^F$$
$$\Longrightarrow (\exists t^R,\ \exists s^{R'}\ \in S^R\ .\ s^{R'} = AF(s^{F'}) \text{ and } \langle s^R, t^R, s^{R'}\rangle\ \in\ R^R{}_q)$$

Proof:

We develop a proof by induction on the length of the execution sequence.

1. Basis:

$AF(s^F{}_0) = s^R{}_0$

Trivially true by definition of $s^F{}_0$ and $s^R{}_0$.

2. Induction Step:

We do a case analysis on the type of the transition t^F:

- $t^F$is not an appending transition

  We know that $f^F(c^F, s^F) = \mathsf{true}$, $g^F(u^F, s^F) = s^{F'}$ and $s^R = AF(s^F)$ and we want to prove that $f^R(c^R, s^R) = \mathsf{true}$, $g^R(u^R, s^R) = s^{R'}$ and $s^{R'} = AF(s^{F'})$.

  We choose an $s^F \in S^F$ and an $s^R\ \in S^R\ .\ s^R\ = AF(s^F)$.

  We also choose an $s^{F'}\ .\ g^R(u^R, s^R) = s^{R'}$.

  From **Lemma 4** we have directly that $f^F(c^F, s^F) = f^R(c^R, s^R)$ and therefore $f^R(c^R, s^R) = \mathsf{true}$.

  We are left to prove that $s^{R'} = AF(s^{F'})$.

  From the definition of AF:

$$AF(s^{F'}) = s^{F'}(v, n) \tag{5.47}$$

From the hypothesis:

$$s^{F'}(v,n) = g^F(u^F, s^F)(v,n) \tag{5.48}$$

We chose

$$s^{F'} \ . \ g^R(u^R, s^R) = s^{R'} \tag{5.49}$$

From 5.47, 5.48 and 5.49, what we have to prove becomes

$$\forall v \in Vars \ g^R(u^R, s^R)(v,n) = g^F(u^F, s^F)(v,n)$$

There are two cases:

- If $v \notin update(u^R, s^R)$

  From the definition of g^R:

  $$g^R(u^R, s^R)(v,n) = s^R(v,n)$$

  From the definition of g^F:

  $$g^F(u^F, s^F)(v,n) = s^F(v,n)$$

  But from hypothesis we know that $s^R = AF(s^F)$. From the definition of AF this means:

  $$\forall (v,n) \ s^R(v,n) = s^F(v,n)$$

  Therefore

  $$\forall v \in Vars \ g^R(u^R, s^R)(v,n) = g^F(u^F, s^F)(v,n)$$

- If $v \in update(u^R, s^R)$

  From the definition of g^R:

  $$g^R(u^R, s^R)(v,n) = f^R({e^R}_i, s^R)$$

  From the definition of g^F:

  $$g^F(u^F, s^F)(v,n) = f^F({e^F}_i, s^F)$$

  This is trivially true from **Lemma 4**.

- t^F is an appending transition

  We know that $f^F(c^F \wedge finalLengthOK(s^F, T^F, \text{currentPath}), s^F) = \mathsf{true}$, $g^F(u^F, s^F) = s^{F'}$ and $s^R = AF(s^F)$ and we want to prove that $f^R(c^R, s^R) = \mathsf{true}$, $g^R(u^R, s^R) = s^{R'}$ and $s^{R'} = AF(s^{F'})$.

  But our hypothesis is strictly stronger than the hypothesis of the previous case, when t^F was not an appending transition.

  This implies that we can conclude directly the result of the previous case, which is identical to what we want to prove. We therefore conclude that

$$\forall v \in Vars\ g^R(u^R, s^R)(v, n) = g^F(u^F, s^F)(v, n)$$

5.5.2 Non-termination

We want to prove that FINIT preserves the non-termination property of the extended system consisting of RELAX and the user-given queue lengths. To be more exact, we want to prove that if from any state s of RELAX there exists an external execution step that can be taken, then there exists an external execution step in FINIT that can be taken from any state that maps to s using AF s.t. the system does not go into an ERROR state. We also want to prove that FINIT goes into an ERROR state iff the only transition from the state s would overflow at least one of the queues.

Formally, we want to prove that

$$\begin{aligned}
&\forall s^R \in S^R,\ \forall s^F \in\ S^F, \\
&\quad \text{if } \exists t^R\ .\ AF(s^F) = s^R \text{ and } \langle s^R,\ t^R,\ s^{R'} \rangle\ \in\ {R^R}_q \text{ and } ex^R(t^R) = \mathsf{true} \\
&\quad \Longrightarrow \exists t^F,\ \exists s^{F'} \in S^F\ . \\
&\quad \langle s^F,\ t^F,\ s^{F'} \rangle\ \in\ R^F \text{ and } ex^F(t^F) = \mathsf{true} \text{ and } s^{F'} \neq s_{ERROR} \qquad (5.50)
\end{aligned}$$

Also

$$(\forall s^R \in S^R, \forall s^F \in S^F, \forall t^R \;.$$
$$AF(s^F) = s^R \text{ and } \langle s^R,\ t^R,\ s^{R'} \rangle \in R^R \text{ and } ex^R(t^R) = \mathsf{true}) \text{ we have that}$$
$$\langle s^R,\ t^R,\ s^{R'} \rangle \notin R^R{}_q \text{iff } finalLengthOK(s^F, T^F, \text{currentPath}) = \mathsf{false} \quad (5.51)$$

Our global scheduling algorithm generates a correct transformed specification that correctly deadlocks if the designer specifies lengths for the queues that are not large enough for the particular application. It is the designer's responsability to know what queue length values are enough for the given circuit not to deadlock in practice; we prove that given such lengths, our scheduling algorithm does not introduce deadlock in the system.

Lemma 5: $\forall s^R \in S^R, \forall s^F \in S^F \;.\; AF(s^F) = s^R$ we have that

$$\forall q \in Queues,\ room^R(q, s^R) = room^F(q, s^F)$$

Proof: From the definition of the abstraction function

$$FUNC\ AF^F(s^F) \rightarrow \{\forall v \in Vars \mid s^F(v, n) = s^R(v, n)\}$$

we have that $\forall q \in Queues,\ s^R(q, n) = s^F(q, n)$. Therefore

$$\forall q \in Queues,\ room^R(q, s^R) = room^F(q, s^F)$$

Lemma 6:

$$\forall s^R \in S^R, \forall s^F \in S^F, \forall t^R \;.$$
$$AF(s^F) = s^R \text{ and } \langle s^R,\ t^R,\ s^{R'} \rangle \in R^R \text{ and } ex^R(t^R) = \mathsf{true}$$
$$\exists t^F,\ \exists s^{F'} \in S^F \;.$$
$$\langle s^F,\ t^F,\ s^{F'} \rangle \in R^F \text{ and } ex^F(t^F) = \mathsf{true} \text{ and } u^R = u^F \text{ and } c^R = c^F$$

Proof: By construction of FINIT.

Non-termination Proof:

The proof for 5.50 proceeds by contradiction.

We choose $s^R \in S^R$. We also choose $s^F \in S^F$. $AF(s^F) = s^R$.

If $\exists t^R$. $\langle s^R,\ t^R,\ s^{R'} \rangle \ \in \ {R^R}_q$ and $ex^R(t^R) = \mathsf{true}$ then we know that

$$f^R(c^R, s^R) = \mathsf{true} \tag{5.52}$$

$$g^R(u^R, s^R) = s^{R'} \tag{5.53}$$

$$u^R \Rightarrow \text{overflow}({s^R}_i) \text{ is } \mathsf{false} \tag{5.54}$$

$$ex^R(t^R) = \mathsf{true} \tag{5.55}$$

where ${s^R}_i$ is defined as in 5.46.

We want to prove that $\exists t^F,\ \exists s^{F'} \in S^F$. $\langle s^F,\ t^F,\ s^{F'} \rangle \ \in \ R^F$ and $ex^F(t^F) = \mathsf{true}$ and $s^{F'} \neq s_{ERROR}$

Assume there exists no $s^{F'}$ and t^F s.t.

$$\langle s^F,\ t^F,\ s^{F'} \rangle \ \in \ R^F \text{ and } ex^F(t^F) = \mathsf{true} \text{ and } s^{F'} \neq s_{ERROR} \tag{5.56}$$

We know from the construction of the transition system that none of the internal transitions affects the content of the queues. Therefore 5.56 implies that

$$\forall q \ \in \ Queues,\ room^F(q, {s^F}_{|T|+1}) = room^F(q, s^F) \tag{5.57}$$

From 5.54 we have that $\forall q \ \in \ Queues,\ length(q, {s^R}_{|T|+1}) \leq maxLength(q)$, which is equivalent to

$$\forall q \ \in \ Queues,\ room^R(q, {s^R}_{|T|+1}) \geq 0 \tag{5.58}$$

- If t^R is an appending transition

  We choose $s^{F'} = g^F(u^F, s^F)$. We want to prove that $f^F(c^F, s^F) = \mathsf{true}$.

  We know that $AF(s^F) = s^R$. From **Lemma 4** we have directly that $f^F(c^F, s^F) = f^R(c^R, s^R)$. From 5.52 we infer that $f^F(c^F, s^F) = \mathsf{true}$.

From the construction of the transition system we have that if t^R is an appending transition in ${R^R}_q$, then t^F is an appending transition in R^F and the reverse, i.e.

$$append(t^R, q) = \mathsf{true} \text{ iff } append(t^F, q) = \mathsf{true} \tag{5.59}$$

Also from the construction of the transition system we know that all appending transitions are external, i.e. $ex^F(t^F) = \mathsf{true}$.

Because t^R is an appending transition, from 5.58 we have that for each $q \in Queues$ s.t. $append(t^R, q) = \mathsf{true}$, $room^R(q, s^R) > 0$. From the **Lemma 5** we have that

$$\forall q \in Queues,\ room^F(q, s^F) > 0 \tag{5.60}$$

From 5.57 and 5.60 we infer that $\forall q \in Queues\ .\ append(t^R, q) = \mathsf{true}$ and we have that $room^F(q, {s^F}_{|T|+1}) > 0$.

But from 5.59 $append(t^R, q) = \mathsf{true}$ iff $append(t^F, q) = \mathsf{true}$.

This implies that $finalLengthOK(s^F, T^F, \text{currentPath}) = \mathsf{true}$.

– If $t^R$is not an appending transition

We choose ${s^F}' = g^F(u^F, s^F)$. We want to prove that $f^F(c^F, s^F) = \mathsf{true}$.

We know that $AF(s^F) = s^R$. From **Lemma 4** we have directly that $f^F(c^F, s^F) = f^R(c^R, s^R)$. From (1) we infer that $f^F(c^F, s^F) = \mathsf{true}$.

By construction of $T^F$from T^R, $t^F$is an external transition iff $t^R$is an external transition. From (4) we infer that $ex^F(t^F) = \mathsf{true}$.

This concludes the first part of the non-termination proof.

We now want to prove the second part of the non-termination theorem, which says that:

$(\forall s^R \in S^R, \forall s^F \in S^F, \forall t^R\ .$

$AF(s^F) = s^R$ and $\langle s^R,\ t^R,\ s^{R'}\rangle\ \in\ R^R$ and $ex^R(t^R) = \mathsf{true}$) we have that

$\langle s^R,\ t^R,\ s^{R'}\rangle\ \notin\ {R^R}_q$ iff

$\exists t^F,\ \exists s^{F'} \in S^F\ .\ \langle s^F,\ t^F,\ s^{F'}\rangle\ \in\ R^F$ and $ex^F(t^F) = \mathsf{true}$ and

$$finalLengthOK(s^F, T^F, \text{currentPath}) = \mathsf{false} \tag{5.61}$$

For each $s^R \in S^R$, and $\forall s^F\ \in\ S^F\ .\ AF(s^F) = s^R$, we choose some t^R s.t. $\langle s^R,\ t^R,\ s^{R'}\rangle\ \in\ R^R$ and $ex^R(t^R) = \mathsf{true}$.

If t^R is not an appending transition, then $\langle s^R,\ t^R,\ s^{R'}\rangle\ \in\ {R^R}_q$ and the function $finalLengthOK$ is not defined. There is nothing to prove in this case.

If t^R is an appending transition, we have the following:

$\langle s^R,\ t^R,\ s^{R'}\rangle\ \notin\ {R^R}_q$ is equivalent to saying that $u^R \Rightarrow \text{overflow}({s^R}_i)$, where ${s^R}_i$ is defined as in 5.46. This means

$$\exists q_k \in Queues\ .\ u^R \Rightarrow length(q_k, {s^R}_{|T|+1}) > maxLength(q_k)$$

If we write $length(q_k, {s^R}_{|T|+1})$ as $maxLength(q_k) - room^R(q_k, {s^R}_{|T|+1})$ then we have that $\langle s^R,\ t^R,\ s^{R'}\rangle\ \notin\ {R^R}_q$ is equivalent to

$$\exists q_k \in Queues\ .\ u^R \Rightarrow room^R(q_k, {s^R}_{|T|+1}) < 0 \tag{5.62}$$

From the definition of $finalLengthOK(s^F, T^F, \text{currentPath})$, we have that

$$finalLengthOK(s^F, T^F, \text{currentPath}) = \mathsf{false}$$

is equivalent to

$$\exists q \in Queues\ .\ finalLengthOKqueue(s^F, T^F, q, \text{currentPath}) = \mathsf{false}$$

From the definition of $finalLengthOKqueue(s^F, T^F, q, \text{currentPath})$ we have that

$$\exists q \in Queues\ .$$
$$\sum_i eval(tail(t_i, q) \bigwedge (f^F({c^F}_i \wedge finalLengthOK({s^F}_i, T^F, newPath), {s^F}_i)$$
$$\vee (t_i \in \text{currentPath}))) +$$
$$room^F(q, s^F) \leq 0,\ \forall i \in \{l^F + 1, ..., |T|\}\ .\ tail(t_i, q) = \mathsf{true},$$

where ${s^F}_{l^F} = s^F$ and

$${s^F}_i = \begin{cases} g^F({u^F}_{i-1}, {s^F}_{i-1}) & \text{if } f^F({c^F}'_{i-1}, {s^F}_{i-1}) \\ {s^F}_{i-1} & \text{otherwise} \end{cases}$$

$$newPath = \begin{cases} \text{currentPath} & \text{if } t_i \in \text{currentPath} \\ \text{currentPath} \vee t_i & \text{otherwise} \end{cases}$$

Therefore,

$$\exists t^F,\ \exists {s^F}' \in S^F\ .\ \langle s^F,\ t^F,\ {s^F}' \rangle\ \in\ R^F \text{ and } ex^F(t^F) = \mathsf{true} \text{ and}$$

$$finalLengthOK(s^F, T^F, \text{currentPath}) = \mathsf{false}$$

is equivalent to

$$\exists t^F,\ \exists {s^F}' \in S^F,\ \exists q \in Queues\ .\ \langle s^F,\ t^F,\ {s^F}' \rangle\ \in\ R^F \text{ and } ex^F(t^F) = \mathsf{true} \text{ and}$$
$$(u^R \Rightarrow \sum_i eval(tail(t_i, q) \bigwedge (f^F({c^F}_i \wedge finalLengthOK({s^F}_i, T^F, newPath), {s^F}_i)$$
$$\vee (t_i \in \text{currentPath}))) + room^F(q, s^F) < 0)$$

which is the same as

$$\exists t^F,\ \exists {s^F}' \in S^F,\ \exists q \in Queues\ .\ \langle s^F,\ t^F,\ {s^F}' \rangle\ \in\ R^F \text{ and } ex^F(t^F) = \mathsf{true} \text{ and}$$
$$(u^R \Rightarrow \sum_i eval(tail(t_i, q) \bigwedge (f^F({c^F}_i \wedge finalLengthOK({s^F}_i, T^F, newPath), {s^F}_i)$$
$$\vee (t_i \in \text{currentPath}))) + room^F(q, {s^F}_{|T|+1}) < 0)$$

If the rule graph is acyclic, this expression is the same as writing that

$$\exists t^F,\ \exists s^{F'} \in S^F,\ \exists q \in Queues\ .\ \langle s^F,\ t^F,\ s^{F'} \rangle\ \in\ R^F \text{ and}$$
$$ex^F(t^F) = \mathsf{true} \text{ and } (u^R \Rightarrow room^F(q, s^F{}_{|T|+1}) < 0) \qquad (5.63)$$

If the rule graph is cyclic, then because

$$room^F(q, s^F{}_{|T|+1}) \leq room^F(q, s^F{}_{|T|+1}) +$$
$$\sum_i eval(tail(t_i, q) \bigwedge (t_i \in \text{currentPath}) \bigwedge$$
$$\overline{f^F(c^F{}_i \wedge finalLengthOK(s^F{}_i, T^F, newPath), s^F{}_i))}$$

we obtain the same inequality as in the acyclic case, i.e.

$$\exists t^F,\ \exists s^{F'} \in S^F,\ \exists q \in Queues\ .\ \langle s^F,\ t^F,\ s^{F'} \rangle\ \in\ R^F \text{ and } ex^F(t^F) = \mathsf{true} \text{ and}$$
$$(u^R \Rightarrow room^F(q, s^F{}_{|T|+1}) < 0) \qquad (5.64)$$

From 5.61, 5.62, 5.63, and 5.64 we infer that what we have to prove is

$$\exists t^F \exists s^{F'} \in S^F\ .\ \langle s^F,\ t^F,\ s^{F'} \rangle\ \in\ R^F \text{ and } ex^F(t^F) = \mathsf{true} \text{ and}$$

$$\exists q_k \in Queues\ .\ (u^R \Rightarrow room^R(q_k, s^R{}_{|T|+1}) < 0) \text{ iff}$$

$$\exists q \in Queues\ .\ (u^F \Rightarrow room^F(q, s^F{}_{|T|+1}) < 0) \qquad (5.65)$$

From **Lemma 5** we have that

$$room^R(q_k, s^R) = room^F(q_k, s^F) \qquad (5.66)$$

We also have the following equivalences:

$$\exists q_k \in Queues\ .\ (u^R \Rightarrow room^R(q_k, s^R{}_{|T|+1}) < 0) \text{ is equivalent to}$$

$$\exists q_k \in Queues\ .$$
$$u^R \Rightarrow room^R(q_k, s^R) - nrFollowingApp(t^R) + nrFollowingRm(t^R) < 0 \qquad (5.67)$$

$\exists q \in Queues$. $(u^F \Rightarrow room^F(q, {s^F}_{|T|+1}) < 0)$ is equivalent to

$\exists q \in Queues$.

$$u^F \Rightarrow room^F(q, s^F) - nrFollowingApp(t^F) + nrFollowingRm(t^F) < 0 \quad (5.68)$$

From **Lemma 6** we have that

$$u^R = u^F \text{ and } c^R = c^F \quad (5.69)$$

To prove 5.65, from 5.66, 5.67, 5.68, 5.69 and for $q = q_k$, we deduce that we only have to prove that corresponding transitions in R^R and R^F, following t^R and t^F, either both execute, or neither does.

After t^R has executed from state s^R, and t^F has executed from state s^F, because $AF(s^F) = s^R$ and 5.69, we can infer that $AF(s^{F'}) = s^{R'}$.

Let's look now at the transitions following t^R and t^F, respectively. Let's call the immediately following new corresponding transitions $t^{R,}$ and $t^{F,}$.

If $t^{R,}$ and $t^{F,}$ are not appending transitions, they can execute, provided that their enabling conditions are satisfied. Because $AF(s^{F'}) = s^{R'}$, from **Lemma 6** we have that $u^{R'} = u^{F'}$ and $c^{R'} = c^{F'}$. After execution, the new system states, $s^{R"}$ and $s^{F"}$, are again in the relation $AF(s^{F''}) = s^{R''}$.

If $t^{R,}$ and $t^{F,}$ are appending transitions, then we reduced proving 5.61 for t^R and t^F to proving 5.61 for $t^{R,}$ and $t^{F,}$. Because the number of transitions following t^R and t^F, correspondingly, is finite, we will eventually reach the last transitions in each RELAX and FINIT, in states ${s^R}_{|T|+1}$ and ${s^F}_{|T|+1}$. Here 5.61 holds, since there are no more following transitions. This concludes the non-termination proof.

5.5.3 Correctness for Groups of Transitions

For cyclic specifications, the algorithm considers the coordinated execution of groups of transitions, rather than transitions in isolation. We call a **phase** a sequence of external transition executions, such that each transition executes at most once.

The simulation theorem states that for any phase in FINIT, we can find a phase in RELAX with the same execution sequence. The proof is virtually identical to the simulation proof for one transition, and is based on the fact that the enabling condition of each of the transitions is strictly stronger in FINIT than in RELAX.

We can formulate a new non-termination theorem for groups of transitions which states that, if from any state s of RELAX there exists a phase that takes the system into a new state in which none of the queues overflows its designer-specified length, then from any state in FINIT, s', that maps to s using AF, there exists a phase in FINIT which does not take the new system into an ERROR state. The proof goes by contradiction and works on a phase instead of a single transition at a time. The idea is to infer that the transition from s' in the FINIT phase corresponding to the executing RELAX phase would have its enabling condition satisfied, and therefore execute.

Chapter 6

Experimenting with the System

We have implemented a synthesis system based on the algorithms presented in this thesis. This system produces synthesizable Verilog implementations at the RTL level. To evaluate our system, we developed a set of benchmarks in our specification language and used our system to produce hardware implementations of these applications. This benchmark set includes both a processor application and a few standard DSP applications. We performed a set of experiments designed to investigate two aspects of using our system: (1) how natural and concise it is for the designer to write circuit specifications in our language, and (2) how well the resulting implementations perform.

The rest of the chapter is structured as follows. The first section describes our benchmark applications and provides a complete specification for each benchmark. Next we present the steps of the performance evaluation process. The third section gives area and clock cycle numbers for our benchmarks and functionally equivalent versions written manually in Verilog. Section 6.4 comments and gives some qualitative numbers relative to the design effort, while Section 6.5 makes the performance evaluation. We summarize in Section 6.6.

6.1 The Applications

We selected the following set of benchmarks: a circuit that performs bubble-sort, a butterfly network, a cascaded FIR filter and a processor. The remaining of this section contains one subsection for each of our benchmark applications, and presents complete specifications for these benchmarks.

6.1.1 Bubblesort Sorting Network

Figure 6.2 below presents the specification of the bubblesort network for six 8-bit integers in Figure 6.1. We are implementing a variant of bubblesort called odd-even transposition. For `n` even, this algorithm sorts `n` elements in `n` phases, each of which requiring `n/2` compare-exchange(CE) operations. The algorithm alternates between two phases, the odd phase and the even phase. The odd phase compares each odd index number with its following even index number and swaps them if they are out of sequence. The even phase compares each even index number with its following odd index number and swaps them if they are out of sequence. After `n/2` pairs of odd-even phases, the numbers are sorted.

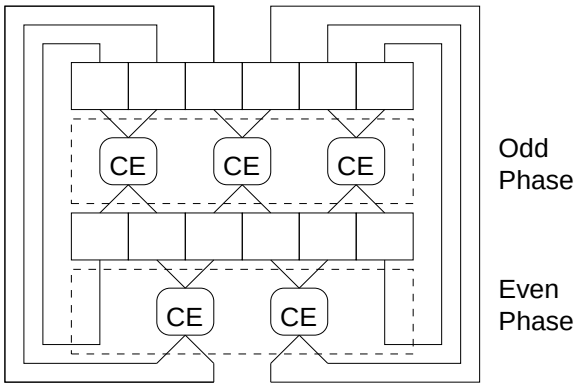

Figure 6.1: Bubble Sort for six numbers

In Figure 6.2, the numbers are given as an array `nr` of integers. Indices into this

```
state nr:  int[6];
state q1, q2, q3, q4, q5, q6:  queue(int);

// Odd Phase Module
1:  true ->
        q1 = (nr[1] > nr[2])?  append(q1,nr[1]):append(q1,nr[2]),
        q2 = (nr[1] > nr[2])?  append(q2,nr[2]):append(q2,nr[1]),
        q3 = (nr[3] > nr[4])?  append(q3,nr[4]):append(q3,nr[3]),
        q4 = (nr[3] > nr[4])?  append(q4,nr[3]):append(q4,nr[4]),
        q5 = (nr[5] > nr[6])?  append(q5,nr[6]):append(q5,nr[5]),
        q6 = (nr[5] > nr[6])?  append(q6,nr[5]):append(q6,nr[6]);

// Even Phase Module
2:  u = head(q2) and v = head(q3) ->
        nr[2] = (u < v)?  u:v,
        nr[3] = (u < v)?  v:u,
        q2 = tail(q2),
        q3 = tail(q3);
3:  u = head(q4) and v = head(q5) ->
         nr[4] = (u > v)?  v:u,
         nr[5] = (u > v)?  u:v,
         q4 = tail(q4),
         q5 = tail(q5);
4:  v = head(q1) ->
         nr[1] = v,
         q1 = tail(q1);
5:  v = head(q6) ->
         nr[6] = v,
         q6 = tail(q6);
```

Figure 6.2: Bubblesort Network for six 8-bit integers

array start at 1. One of the most natural ways to describe such a function in a modular fashion is to have one module for each of the two phases. There are six other locations q_0 to q_5 that hold integers representing the intermediate values after the odd phase. We represent them as FIFO queues of integer type. The first module contains only rule `1`. Depending on the result of the comparisons, each number in `nr` is inserted in one of the output queues q_0 to q_5. The second module, consisting of rules `2` to `5`, reads these numbers from the queues, executes the even-odd compare-exchange operations and writes the results back into the array `nr`, then removes the values it read from the queues.

For applications that are very regular in structure, having meta-programming support to parameterize and iterate would make specifications more concise and elegant. The basic operation that we need is to generate multiple instances of a rule. For n even, the new specification is given in Figure 6.3.

6.1.2 Butterfly Sorting Network

The butterfly configuration is used in bitonic sorting networks and FFTs. A bitonic sorting network recursively converts an input sequence into shorter bitonic sequences, until we obtain subsequences of size one. At that point, the output is sorted in monotonically increasing order. A bitonic sequence is a sequence of elements $\langle a_0, \ldots, a_{n-1} \rangle$ with the property that either (1) there exists an index `i, 0<=i<=n-1`, such that $\langle a_0, \ldots, a_i \rangle$ is monotonically increasing and $\langle a_{i+1}, \ldots, a_{n-1} \rangle$ is monotonically decreasing, or (2) there exists a cyclic shift of indices so that (1) is satisfied.

Figure 6.4 shows the bitonic sort for a sequence of four numbers. `BM[n]` denotes a bitonic merging network of input size `n`. The sign before `BM[n]` denotes that the network uses either `<` comparators, for `+`, or `>` comparators, for `-`. In our figure, the algorithm first converts the input sequence into a bitonic sequence of size 2; it does this by passing the upper two numbers through a `+BM[2]`

```
state nr:  int[n];
state q_1, ...  q_n:  queue(int);

// Odd Phase Module
1:  for(i=0; i<=n/2-1; )
  true ->
        q_{2*i+1}= (nr[2*i+1] > nr[2*i+2])?
                        append(q_{2*i+1},nr[2*i+2]):append(q_{2*i+1},nr[2*i+1]),
        q_{2*i+2}= (nr[2*i+1] > nr[2*i+2])?
                        append(q_{2*i+2},nr[2*i+1]):append(q_{2*i+2},nr[2*i+2]);

// Even Phase Module
2:  for(i=1; i<=n/2-1; )
  u = head(q_{2*i}) and v = head(q_{2*i+1}) ->
        nr[2*i] = (u < v)?  u:v,
        nr[2*i+1] = (u < v)?  v:u,
        q_{2*i+1} = tail(q_{2*i+1}),
        q_{2*i} = tail(q_{2*i});
3:  v = head(q_1) ->
         nr[1] = v,
         q_1 = tail(q_1);
4:  v = head(q_n) ->
         nr[n] = v,
         q_n = tail(q_n);
```

Figure 6.3: Bubblesort Network with Meta-Programming Support

network and the lower two through a `-BM[2]` network. The second and last step of the algorithm is to convert the newly obtained sequence into an output bitonic sequence of size 1. This operation is performed by `+BM[4]`.

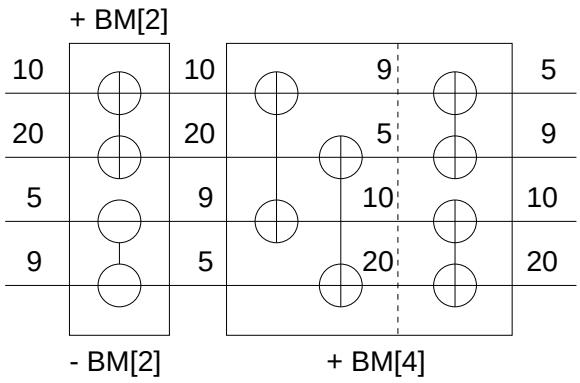

Figure 6.4: Bitonic Sort for four numbers

Figure 6.5 gives the specification for the configuration in Figure 6.4. The idea is the following. We want to have one module for each recurrent pattern in the network. Figure 6.4 displays two patterns. The first and third column compare the elements in the sequence at distance 1 from each other; this is our first configuration pattern and will be specified by one module. The second column compares the elements in the sequence at distance 2 from each other and it will be described by another module. Let's use the letter `p` for the distance between the elements we compare and $\texttt{M}_p$ for the corresponding module. We want module $\texttt{M}_p$ to execute the desired comparisons only if (1) $\texttt{M}_{p+1}$ within the same $\texttt{BM[2}^k\texttt{]}$ already executed, if `p != k` or (2) $\texttt{BM[2}^{k-1}\texttt{]}$ finished executing, if if `p == k`. Therefore, each module first tests whether `k >= p` and only if this is true performs its function; otherwise just passes the values unchanged from the input to the output queue. Within each module, `>` and `<` comparisons alternate in groups of 2^{k-1}; we can see this in rule `4`. `k` starts at `1`.

For 2^n numbers, $n \geq 2$, Figure 6.6 shows the specification of a bitonic sort network using meta-programming support. The division of rules in modules is not shown explicitly for reasons of conciseness of the specification.

```
state nr:  int[4];
state k:  int;
state q1, q2, q3, q4:  queue(int);

// Module to compare the elements at distance 2 from each other,
// of the comparisons required by the algorithm
1:  k >= 2 ->
    q1 = (nr[1] > nr[3])?  append(q1,nr[3]):append(q1,nr[1]),
    q3 = (nr[1] > nr[3])?  append(q3,nr[1]):append(q3,nr[3]),
    q2 = (nr[2] > nr[4])?  append(q2,nr[4]):append(q2,nr[2]),
    q4 = (nr[2] > nr[4])?  append(q4,nr[2]):append(q4,nr[4]);
2:  k < 2 ->
    q1 = append(q1,nr[1]),
    q2 = append(q2,nr[2]),
    q3 = append(q3,nr[3]),
    q4 = append(q4,nr[4]),

// Module to compare the elements at distance 1 from each other,
// of the comparisons required by the algorithm
3:  k >= 1 and a1 = head(q1) and a2 = head(q2) ->
    nr[1] = (a1 > a2)?  a2:a1,
    nr[2] = (a1 > a2)?  a1:a2,
    q1 = tail(q1),
    q2 = tail(q2);
4:  k == 1 and a3 = head(q3) and a4 = head(q4) ->
    nr[3] = (a3 > a4)?  a3:a4,
    nr[4] = (a3 > a4)?  a4:a3,
    q3 = tail(q3),
    q4 = tail(q4);
5:  k == 2 and a3 = head(q3) and a4 = head(q4) ->
    nr[3] = (a3 > a4)?  a4:a3,
    nr[4] = (a3 > a4)?  a3:a4,
    q3 = tail(q3),
    q4 = tail(q4);

6:  true -> k = k + 1;
7:  k == 3 ->
    k = 1;
```

Figure 6.5: Butterfly Sorting Network

```
// Sorting 2^n numbers, with n ≥ 2
// k is the power of 2 in BM[2^k], p = 2^pp is the sorting distance
1:  for (k = 1; k ≤ n; )
      for (pp = 0; pp ≤ k-1; )
        k < 2^pp → for (i = 1; i ≤ 2^n; ) q_i= append(q_i, nr[i]);
2:  for (k = 2; k ≤ n; )
      for (pp = 0; pp ≤ k-1; )
        // nr_loops = nr_comparators / nr_elements_in_loop = 2^(n-1) / 2^k = 2^(n-k-1)
        for (j = 0; j < 2^(n-k-1); )
          k ≥ 2^pp →
              for (l = 4*j*(k-1) + 1; l ≤ 2*(k-1)*(2*j+1); )
                  q_l = (nr[l]>nr[l+2^pp]) ?
                        append(q_l,nr[l+2^pp]):append(q_l,nr[l]),
                  q_{l+2^pp} = (nr[l]>nr[l+2^pp]) ?
                        append(q_{l+2^pp},nr[l]):append(q_{l+2^pp},nr[l+2^pp]);
              for (l = 2*(2*j+1)*(k-1) + 1; l ≤ 4*(k-1)*(j+1); )
                  q_l = (nr[l] ≤ nr[l+2^pp]) ?
                        append(q_l,nr[l+2^pp]):append(q_l,nr[l]),
                  q_{l+2^pp} = (nr[l] ≤ nr[l+2^pp]) ?
                        append(q_{l+2^pp},nr[l]):append(q_{l+2^pp},nr[l+2^pp]);
3:  k = 1 and pp = 0 →
    for (j = 0; j < 2^(n-2); )
      q_{4*j+1} = (nr[4*j+1]<nr[4*j+2]) ?
                        append(q_{4*j+1},nr[4*j+1]):append(q_{4*j+1},nr[4*j+2]),
      q_{4*j+2} = (nr[4*j+1]<nr[4*j+2]) ?
                        append(q_{4*j+2},nr[4*j+2]):append(q_{4*j+2},nr[4*j+1]),
      q_{4*j+3} = (nr[4*j+3] ≥ nr[4*j+4]) ?
                        append(q_{4*j+3},nr[4*j+3]):append(q_{4*j+3},nr[4*j+4]),
      q_{4*j+4} = (nr[4*j+3] ≥ nr[4*j+4]) ?
                        append(q_{4*j+4},nr[4*j+4]):append(q_{4*j+4},nr[4*j+3]);
```

Figure 6.6: Butterfly Sorting Network with Meta-Programming Support

6.1.3 Cascaded FIR

Figure 6.7 shows the direct form of a digital FIR filter. The filter produces an output, y_n, that is the weighted sum of the current and past inputs, x_n; b_i are the coefficients.

$$y_n = b_0 x_n + b_1 x_{n-1} + b_2 x_{n-2} + \ldots + b_q x_{n-q} = \sum_{j=0}^{q} b_j x_{n-j}$$

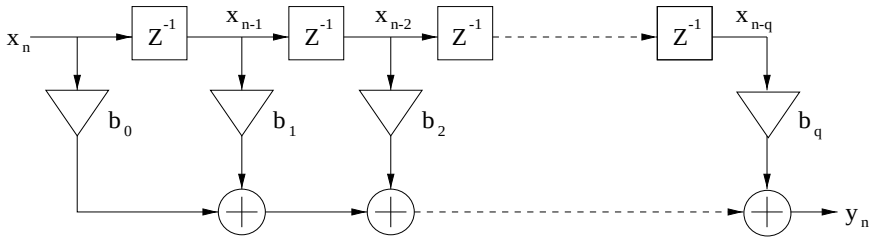

Figure 6.7: The direct form for a digital filter

The z^{-1} block in Figure 6.7 represents the unit delay operator. Another equivalent canonical form for the FIR filter above is the cascade form presented in Figure 6.8. In practice, because the accuracy requirements on the coefficients are often severe for the direct form, the designers favor the use of the cascade form. A cascaded FIR filter of order `2*c` is characterized by a transfer function `H(z)` given by

$$H(z) = h[0] * \Pi_{k=1}^{c}(1 + \beta_{1k} z^{-1} + \beta_{2k} z^{-2}), \text{ with } \beta_{2(2*c)} = 0$$

Figure 6.9 specifies such a cascaded FIR filter with `2*c` coefficients, using meta-programming support. `Mup` contains the coefficients $\beta_{1k}, k \in \{1, ..., c\}$; `Mdown` contains the coefficients $\beta_{2k}, k \in \{1, ..., c\}$. The input vector `in` is size `N` long. $h[0]$ is initially `in[1]`. The specification is a direct translation of the operations involved in computing the transfer function. Our benchmark implements a cascaded FIR of order 12 (with 12 coefficients).

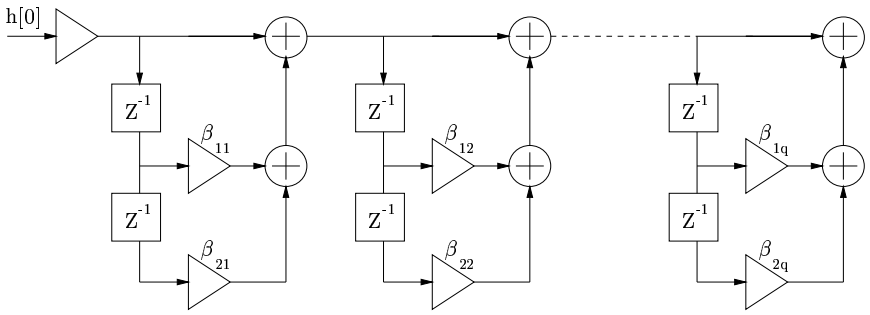

Figure 6.8: The cascade form for a digital filter

```
state in:  int[N];
state out, cnt:  int;
state Mup, Mdown :  int[c];
state q11,...,q1c,q21,...,q2c :  queue(int);
1:  for(i=1; i<= c; )
1<= cnt <= N ->
        q1i = append(q1i,Mup[i]*in[cnt]+1);
2:  for(i=1; i<= c; )
cnt = 1 and ei = head(q1i) ->
        q2i = append(q2i,ei),
        q1i = tail(q1i);
3:  for(i=1; i<= c; )
1 < cnt <= N and ei = head(q1i) ->
        q2i = append(q2i,ei+Mdown[i]*in[cnt-1]),
        q1i = tail(q1i);
4:  m1 = head(q21) and ...  and mc = head(q2c) ->
        out = m1*...*mc*in[1],
        q21 = tail(q21),...
        q2c = tail(q2c),
        cnt = cnt+1;
```

Figure 6.9: Cascaded FIR Filter of Order `2*c` with Meta-Programming Support

6.1.4 Processor

Figure 6.10 presents the specification of a 32-bit datapath, RISC-style, linearly pipelined processor that supports an instruction set including load, store, jump, ALU, multiply and variable shift operations.

The specifications may become more concise if we introduce macro constructs in our language. For our processor example, we could use the macro defined below, which takes as a parameter a register name:

```
noInstruction(r) = notin(<loadce r _>,eq) and notin(<adde r _>,eq)
  and notin(<sube r _>,eq) and notin(<ande r _>,eq) and notin(<shle
  r _>,eq) and notin(<mule r _>,eq) and notin(<loade r _>,eq)
```

Furthermore, we can define two macros for some of the binary operations in `ins` and their corresponding operations in `exeins` as:

```
  2Op = add | sub | and | or | xor | rhl | shl | mul | load

  2OpE = case(2Op)
          add:  adde
          sub:  sube
          and:  ande
          or:  ore
          xor:  xore
          rhl:  rhle
          shl:  shle
          mul:  mule
          load:  loade
```

Using these macros considerably reduces the size of many enabling conditions, as well as the number of update rules in Figure 6.10, as you can see in Figure 6.11.

```
type reg = int;
type val = int;
type loc = int;
type addr = int;
type R31 = int;
type ins = <loadc r:reg v:val> | <loadpc r:reg> | <load rd:reg ra:reg> |
           <add rd:reg rs:reg> | <sub rd:reg rs:reg> |
           <and rd:reg rs:reg> | <or rd:reg rs:reg> |
           <xor rd:reg rs:reg> | <rhl rd:reg rs:reg> |
           <shl rd:reg rs:reg> | <mul rd:reg rs:reg> |
           <jz rc:reg rl:reg> | <jalr r:reg> | <store ra:reg rv:reg>;
type exeins = <loadce r:reg v:val> | <loade rd:reg a:addr> |
             <adde rd:reg v:val> | <sube rd:reg v:val> |
             <ande rd:reg v:val> | <ore rd:reg v:val> |
             <xore rd:reg v:val> | <rhle rd:reg v:val> |
             <shle rd:reg v:val> | <mule rd:reg v:val> |
             <jze v:val l:loc> | <jalre v:val> | <storee a:addr v:val>;

state im:  ins[];
state dm:  int[];
state rf:  int[];
state pc:  int;
state iq:  queue(ins);
state eq:  queue(exeins);

1:  true ->
    iq = append(iq,im[pc]),
    pc = pc+1;
2:  <loadc r v> = head(iq) ->
    iq = tail(iq),
    eq = append(eq,<loadce r v>)
3:  <loadpc r> = head(iq) ->
    iq = tail(iq),
    eq = append(eq,<loadce r pc>);
```

```
4:  <add rd rs> = head(iq) and notin(<loadce rs _>,eq) and
    notin(<adde rs _>,eq) and notin(<sube rs _>,eq) and
    notin(<ande rs _>,eq) and notin(<shle rs _>,eq) and
    notin(<mule rs _>,eq) and notin(<loade rs _>,eq) ->
    iq = tail(iq),
    eq = append(eq,<adde rd rf[rs]>);
5:  <sub rd rs> = head(iq) and notin(<loadce rs _>,eq) and
    notin(<adde rs _>,eq) and notin(<sube rs _>,eq) and
    notin(<ande rs _>,eq) and notin(<shle rs _>,eq) and
    notin(<mule rs _>,eq) and notin(<loade rs _>,eq) ->
    iq = tail(iq),
    eq = append(eq,<sube rd rf[rs]>);
6:  <and rd rs> = head(iq) and notin(<loadce rs _>,eq) and
    notin(<adde rs _>,eq) and notin(<sube rs _>,eq) and
    notin(<ande rs _>,eq) and notin(<shle rs _>,eq) and
    notin(<mule rs _>,eq) and notin(<loade rs _>,eq) ->
    iq = tail(iq),
    eq = append(eq,<ande rd rf[rs]>);
7:  <or rd rs> = head(iq) and notin(<loadce rs _>,eq) and
    notin(<adde rs _>,eq) and notin(<sube rs _>,eq) and
    notin(<ande rs _>,eq) and notin(<shle rs _>,eq) and
    notin(<mule rs _>,eq) and notin(<loade rs _>,eq) ->
    iq = tail(iq),
    eq = append(eq,<ore rd rf[rs]>);
8:  <xor rd rs> = head(iq) and notin(<loadce rs _>,eq) and
    notin(<adde rs _>,eq) and notin(<sube rs _>,eq) and
    notin(<ande rs _>,eq) and notin(<shle rs _>,eq) and
    notin(<mule rs _>,eq) and notin(<loade rs _>,eq) ->
    iq = tail(iq),
    eq = append(eq,<xore rd rf[rs]>);
9:  <rhl rd rs> = head(iq) and notin(<loadce rs _>,eq) and
    notin(<adde rs _>,eq) and notin(<sube rs _>,eq) and
    notin(<ande rs _>,eq) and notin(<shle rs _>,eq) and
    notin(<mule rs _>,eq) and notin(<loade rs _>,eq) ->
    iq = tail(iq),
    eq = append(eq,<rhle rd rf[rs]>);
```

```
10:  <shl rd rs> = head(iq) and notin(<loadce rs _>,eq) and
    notin(<adde rs _>,eq) and notin(<sube rs _>,eq) and
    notin(<ande rs _>,eq) and notin(<shle rs _>,eq) and
    notin(<mule rs _>,eq) and notin(<loade rs _>,eq) ->
    iq = tail(iq),
    eq = append(eq,<shle rd rf[rs]>);
11:  <jz rc rl> = head(iq) and notin(<loadce rc _>,eq) and
    notin(<adde rc _>,eq) and notin(<sube rc _>,eq) and
    notin(<ande rc _>,eq) and notin(<shle rc _>,eq) and
    notin(<mule rc _>,eq) and notin(<loade rc _>,eq) and
    notin(<loadce rl _>,eq) and notin(<adde rl _>,eq) and
    notin(<sube rl _>,eq) and notin(<ande rl _>,eq) and
    notin(<shle rl _>,eq) and notin(<mule rl _>,eq) and
    notin(<loade rl _>,eq) ->
    iq = tail(iq),
    eq = append(eq,<jze rf[rc] rf[rl]>);
12:  <load rd ra> = head(iq) and notin(<loadce ra _>,eq) and
    notin(<adde ra _>,eq) and notin(<sube ra _>,eq) and
    notin(<ande ra _>,eq) and notin(<shle ra _>,eq) and
    notin(<mule ra _>,eq) and notin(<loade ra _>,eq) ->
    iq = tail(iq),
    eq = append(eq,<loade rd rf[ra]>);
13:  <jalr r> = head(iq) and notin(<loadce r _>,eq) and
    notin(<adde r _>,eq) and notin(<sube r _>,eq) and
    notin(<ande r _>,eq) and notin(<shle r _>,eq) and
    notin(<mule r _>,eq) and notin(<loade r _>,eq) ->
    iq = tail(iq),
    eq = append(eq,<jalre rf[r]>);
14:  <store ra rv> = head(iq) and notin(<loadce ra _>,eq) and
    notin(<adde ra _>,eq) and notin(<sube ra _>,eq) and
    notin(<ande ra _>,eq) and notin(<shle ra _>,eq) and
    notin(<mule ra _>,eq) and notin(<loade ra _>,eq) and
    notin(<loadce rv _>,eq) and notin(<adde rv _>,eq) and
    notin(<sube rv _>,eq) and notin(<ande rv _>,eq) and
    notin(<shle rv _>,eq) and notin(<mule rv _>,eq) and
    notin(<loade rv _>,eq) ->
    iq = tail(iq),
    eq = append(eq,<storee rf[ra] rf[rv]>);
```

```
15:  <mul rd rs> = head(iq) and notin(<loadce rs _>,eq) and
    notin(<adde rs _>,eq) and notin(<sube rs _>,eq) and
    notin(<ande rs _>,eq) and notin(<shle rs _>,eq) and
    notin(<mule rs _>,eq) and notin(<loade rs _>,eq) ->
    iq = tail(iq),
    eq = append(eq,<mule rd rf[rs]>);
16:  <loadce r v> = head(eq) ->
    eq = tail(eq),
    rf = rf[r=v];
17:  <adde rd v> = head(eq) ->
    eq = tail(eq),
    rf = rf[rd = v + rd];
18:  <sube rd v> = head(eq) ->
    eq = tail(eq),
    rf = rf[rd = v - rd];
19:  <ande rd v> = head(eq) ->
    eq = tail(eq),
    rf = rf[rd = v && rd];
20:  <ore rd v> = head(eq) ->
    eq = tail(eq),
    rf = rf[rd = v || rd];
21:  <xore rd v> = head(eq) ->
    eq = tail(eq),
    rf = rf[rd = v xor rd];
22:  <rhle rd v> = head(eq) ->
    eq = tail(eq),
    rf = rf[rd = (rsh(v,rd))];
23:  <shle rd v> = head(eq) ->
    eq = tail(eq),
    rf = rf[rd = (lsh(v,rd))];
24:  <jze x l> = head(eq) and (x==0) ->
    pc = l,
    iq = nil,
    eq = nil;
25:  <jze v l> = head(eq) and (v!=0) ->
    eq = tail(eq);
```

```
26:  <loade rd a> = head(eq) ->
    eq = tail(eq),
    rf = rf[rd=dm[a]];
27:  <jalre v> = head(eq) ->
    eq = tail(eq),
    rf = rf[R31=pc+4],
    pc = v;
28:  <storee a v> = head(eq) ->
    eq = tail(eq),
    dm = dm[a=v];
29:  <mule rd v> = head(eq) ->
    eq = tail(eq),
    rf = rf[rd = v * rd];
```

Figure 6.10: Linear Pipelined Processor

```
1:  true ->
    iq = append(iq,im[pc]),
    pc = pc+1;
2:  <loadc r v> = head(iq) ->
    iq = tail(iq),
    eq = append(eq,<loadce r v>)
3:  <loadpc r> = head(iq) ->
    iq = tail(iq),
    eq = append(eq,<loadce r pc>);
4:  <2Op rd rs> = head(iq) and noInstruction(rs) ->
    iq = tail(iq),
    eq = append(eq,<2OpE rd rf[rs]>);
5:  <jz rc rl> = head(iq) and noInstruction(rc) and noInstruction(rl) ->
    iq = tail(iq),
    eq = append(eq,<jze rf[rc] rf[rl]>);
6:  <jalr r> = head(iq) and noInstruction(r) ->
    iq = tail(iq),
    eq = append(eq,<jalre rf[r]>);
```

```
7:  <store ra rv> = head(iq) and
                              noInstruction(ra) and noInstruction(rv) ->
    iq = tail(iq),
    eq = append(eq,<storee rf[ra] rf[rv]>);
8:  <loadce r v> = head(eq) ->
    eq = tail(eq),
    rf = rf[r=v];
9:  <adde rd v> = head(eq) ->
    eq = tail(eq),
    rf = rf[rd = v + rd];
10:  <sube rd v> = head(eq) ->
    eq = tail(eq),
    rf = rf[rd = v - rd];
11:  <ande rd v> = head(eq) ->
    eq = tail(eq),
    rf = rf[rd = v && rd];
12:  <ore rd v> = head(eq) ->
    eq = tail(eq),
    rf = rf[rd = v || rd];
13:  <xore rd v> = head(eq) ->
    eq = tail(eq),
    rf = rf[rd = v xor rd];
14:  <rhle rd v> = head(eq) ->
    eq = tail(eq),
    rf = rf[rd = (rsh(v,rd))];
15:  <shle rd v> = head(eq) ->
    eq = tail(eq),
    rf = rf[rd = (lsh(v,rd))];
16:  <jze x l> = head(eq) and (x==0) ->
    pc = l,
    iq = nil,
    eq = nil;
17:  <jze v l> = head(eq) and (v!=0) ->
    eq = tail(eq);
18:  <loade rd a> = head(eq) ->
    eq = tail(eq),
    rf = rf[rd=dm[a]];
```

```
19:  <jalre v> = head(eq) ->
     eq = tail(eq),
     rf = rf[R31=pc+4],
     pc = v;
20:  <storee a v> = head(eq) ->
     eq = tail(eq),
     dm = dm[a=v];
21:  <mule rd v> = head(eq) ->
     eq = tail(eq),
     rf = rf[rd = v * rd];
```

Figure 6.11: Linear Pipelined Processor with Meta-Programming Support

6.2 Methodology

To evaluate the performance of our synthesis system for the set of benchmarks that we developed, we follow the steps below.

- Run the original specification through the compiler to obtain a corresponding implementation at the RTL level, in synthesizable Verilog. Simulate the execution of the resulting implementation to test for correct functionality.
- Synthesize the resulting Verilog implementation using the Synopsis Design Compiler to an industry standard .25 micron standard cell process. Gather circuit area and clock cycle numbers.
- Obtain a manually-written Verilog description of the same or functionally equivalent circuit as the one that we are evaluating. Synthesize this specification in the same environment as the automatically generated version. Gather circuit area and clock cycle numbers. This is our reference point for performance evaluation.
- Compare the sets of numbers for the automatic and manually-written versions.

Benchmark	Cycle (MHz)	Area	Register Area
Bubble Sort Network	324.67	1803.75	1371
Butterfly (Bitonic Sort)	204.08	1881.125	969
FIR filter	103.41	7384	3529
RISC Linear Pipelined Processor	88.89	28845	7533

Figure 6.12: Clock Cycle and Area Estimates for Automatically Generated Versions

Architecture	Cycle (MHz)	Area	Register Area
Bubble Sort Network	308.64	1475.75	1192
Butterfly (Bitonic Sort)	120.34	2041.125	1348
FIR filter	—	—	—
SCU RTL 98 DSP	90.91	28359.75	7147

Figure 6.13: Clock Cycle and Area Estimates for Manually Written Versions

We obtained manually written versions of bubblesort and butterfly sort networks from the RAW benchmark suite at MIT. We obtained the processor benchmark off the web, from Santa Clara University.

6.3 Performance Measurements

We tested the generated Verilog for each application using the Cadence NCVerilog simulator. We pushed our benchmarks through the steps presented in Section 6.2. Figure 6.12 and Figure 6.13 show cycle time, total circuit area and non-combinational (register) area numbers for our four benchmarks and the corresponding manually-written Verilog versions.

6.4 Design Effort Evaluation

The goal that we set for our synthesis system from the very beginning was to provide a simple, concise and easy way of specifying circuits without sacrificing the efficiency of the resulting implementation. Of course it is hard to evaluate the design effort because there is really no completely objective measure to quantify it. We are therefore forced to rely on measures like development and testing time, specification size and spec-to-Verilog synthesis time[1] to evaluate our system.

6.4.1 Development Effort

It took us less than five hours to develop the specification for the processor, and about 10 minutes for the other benchmarks. We believe this is significantly faster than developing the corresponding models by hand. Our processor specification contains 13 type and state declarations and 29 rule definitions for module specifications. The SCU RTL 98 DSP application, on the other hand, consists of approximately 885 lines of Verilog code. Our automatically generated implementation consists of about 1200 lines of synthesizable Verilog. The bubblesort benchmark has 2 multiple state declarations and 12 very simple rule definitions. The butterfly network has 3 multiple state declarations and 13 simple rule definitions. The FIR filter benchmark has 5 multiple state declarations and 4 rule definitions. In general, for n being the input vector size, the bubblesort and butterfly benchmarks have $O(n)$ number of rules; for the FIR filter there will also be $O(n)$ rules, for n being the order number of the filter. With meta-programming support, the number of rules gets down to $O(1)$, though the rules are more complex. The manually written specifications have 200 lines of Verilog code for bubblesort and 378 for butterfly.

[1] We call spec-to-Verilog synthesis time the time it takes for our system to synthesize Verilog code from a circuit specification in our language.

6.4.2 Synthesis Time

The spec-to-Verilog synthesis time is roughly proportional to the complexity of the generated control. For all applications except the pipelined processor, our system required less than one minute to generate the Verilog output. For the processor, it took roughly half an hour. We expect spec-to-Verilog synthesis times to get much shorter for circuits with complex control if we replace the existing straightforward resolution mechanism with a more sophisticated automatic deduction mechanism.

The synthesis times for the corresponding automatically generated Verilog versions, and manually written versions compare as follows. For bubblesort, the automated version takes about 1.30 minutes, while the manual version takes about 1.10 minutes to synthesize. For butterfly, the automated version takes about 2.20 minutes, while the manual version takes about 4.30 minutes to synthesize. The automated version for the FIR filter takes about 15.00 minutes to synthesize; we were unable to obtain a free manually written FIR application, and therefore we have no synthesis time to compare against. Our automatically generated RISC processor benchmark takes about 3:17 hours to synthesize; the functionally equivalent, manually developed SCU RTL 98 DSP application takes about 27.00 minutes to synthesize.

6.5 Performance Evaluation

For our first benchmark, the bubblesort network, our compiler automatically generates a circuit that is about 5 percent faster, and about 22 percent larger than the equivalent manually written version. The number of registers generated in the automatically synthesized version is about 15 percent larger than the equivalent number of registers in the manually written application. The extra register area comes from the counters and valid bits associated with each of the pipeline queues. Since the length of each such queue is given by

the designer, the number of extra registers for the automatically generated application does not vary with the number of elements sorted by the bubble-sort network. This means that the larger the number of elements sorted, the closer the gap in the total register area for the automatically generated, and equivalent manually-written versions.

For our second benchmark, we took a manually written version of a bitonic sort network, and we introduced pipeline registers in the same places as in our high-level specification used as source for the automatically generated bitonic sort circuit. After synthesis, the manually written bitonic sort network application yields a circuit that is about 8.5 percent larger, and about 69.59 percent slower, than our automatically generated implementation. This may look like a surprising result, especially since the circuit obtained after introducing the pipeline registers into the manually written application is only about 8.2 percent faster than the original manually written application. At a closer examination, the long clock cycle seems to be due to a chain of NAND, OR, NOR and INVERT gates within the logic in the first pipeline stage of the manually written application. We stress here that we did not specify the same logic for this application in our language, as the one that is coded in the manually written version; rather, we designed and specified the bitonic sort network our own way, keeping the same number of numbers to be sorted, and the same width for the data paths.

We were unable to obtain a free manually developed FIR application to match against our automatically generated FIR circuit. Therefore we only present the clock cycle time and register and total areas for the automatically generated FIR filter version.

In the case of our last, and biggest application, the RISC-style, linear pipelined processor, notice that the synthesized area is roughly the same, while the clock cycle of our processor is within 3 percent of the manually coded version.

In general, inefficiencies come from two sources: the multiplexer chains and the queue inserts. Multiplexer chains appear because our algorithm performs

symbolic execution of all the rules in the specification, and it is unknown at compile-time whether some rule will actually execute during some clock cycle or not. For each state variable, there will be one multiplexer for each rule that can update that variable; chaining multiplexers this way affects both the combinational logic area, and potentially the clock cycle of the resulting implementation. A specification style that makes use of conditionals in the statements as well[2], and specifies rules in such a way that the compiler can infer as many of them to be mutually exclusive as possible, can beneficially reduce the multiplexer chain lengths. Inserts into FIFO queues increase the combinational logic area and potentially the clock cycle of the resulting circuit. This increase is due to implementing additional tests instead of a direct write into a register.

6.6 Summary

Our experience with the synthesis system showed that our approach can deliver circuit implementations that have comparable performance with manually written Verilog versions of the same applications. The main advantages of this approach are that our specification language drastically reduces the development time and effort, and widens the class of people who could write specifications that correctly reflect the designer's intent and build circuits without having to become expert hardware designers. In general, inefficiencies come from the multiplexer chains and the queue inserts.

[2]I.e. when one or more of the state variables updated by a rule need(s) additional condition testing from the other updated variables, use testing within statements rather than splitting the rule in two or more rules.

Chapter 7

Applications: High-level Automatic Pipelining for Sequential Circuits

This chapter presents a new approach for automatically pipelining sequential circuits. All the transformations involved work on a circuit description written in our specification language. The specification style, as well as the language, make pipelining automation a relatively painless task. The algorithm is based on speculation and uses state retention and recovery to respond to incorrect speculations. We also implement two extensions to this basic approach: stalling, which minimizes circuit area by eliminating speculation, and forwarding, which increases the throughput of the generated circuit by forwarding correct values to preceding pipeline stages.

7.1 Overview

Our algorithm starts with a non-pipelined or insufficiently pipelined specification of a circuit and repeatedly shortens its clock cycle by extracting a computation from the critical path and moving it into a new pipeline stage.

The new stage precomputes the result of the selected expression and passes it to the computation of the next stage that uses it. To keep the pipeline full, the new stage must produce the next value of the expression before the final values of the variables it accesses become available. Our algorithm achieves this goal by speculating on the values of these variables. If the speculation is incorrect, the circuit restores its state to match the state before the speculation, flushes the incorrect values from the pipeline, then restarts the computation.

Our algorithm uses several techniques to improve the quality of the pipelined circuit. If the amount of state necessary to recover from an incorrect speculation is excessive, our algorithm can generate stall logic that causes the pipeline stage to stall until the new values are available. This technique eliminates the need for retaining recovery state, as the execution of the pipeline stage will never need to roll back. Our algorithm also generates circuits that forward the correct value to preceding pipeline stages. This technique increases the throughput of the circuit by reducing the amount of time that the circuit spends recovering from incorrect speculations or waiting for correct values to become available.

We have built a prototype implementation of our algorithm. Using our synthesizer as backend, this implementation generates synthesizable Verilog at the RTL level. We have used our implementation to automatically generate pipelined versions of several circuits. Our results show that our automatically generated pipelined circuits are competitive with manually-generated versions.

The remainder of this chapter is structured as follows. Section 7.2 discusses related work. Section 7.3 gives an example of what the starting and derived circuit specifications may look like. Section 7.4 presents the pipelining algorithm. Section 7.5 draws the conclusions.

7.2 Related Work

Many high-level synthesis systems focus on the automatic generation of highly efficient pipelined designs. Most of this work is primarily concerned with functional pipelining. Many synthesis tools target instruction-set architectures[24, 36, 30, 74, 75, 18, 52, 27]; our tool, on the other hand, targets the more general class of sequential circuits. Other approaches start with a C program [51, 42].

Kroening and Paul [56] describe a method of automating the generation of stall and forward logic starting from a given sequential machine. Starting from hardware that is initially partitioned into pipeline stages, the algorithm produces a circuit that can stall in any arbitrary stage while keeping the other stages running, if possible. To implement forwarding, the designer has to specify the registers holding intermediate results that need to be forwarded to previous stages in the pipeline. The goal of our research, in contrast, is to completely automate the pipelining transformation starting from a non-pipelined or insufficiently pipelined specification.

Retiming [23] optimally pipelines combinatorial circuitry. Architectural retiming [47] adds a negative/normal register pair on a latency-constrained path, effectively pipelining the logic without adding latency. It implements the negative register by either precomputation or prediction of the value that it produces.

Research by Hoe and Arvind [50] and Shen and Arvind [82] has introduced an approach to describe, verify and synthesize processors based on term rewriting systems (TRS). They do not implement automatic pipelining, but their specification language offers comparable capabilities in this direction as our language.

7.3 Example

We first present a non-pipelined datapath, then replicate the simple, three-stage, linear pipelined datapath from Chapter 2, which our algorithm can automatically derive from the non-pipelined specification. Section 7.4 shows the intermediate specifications at each step of the algorithm. We chose to present this three-stage linear pipeline because of its simplicity; our algorithm is capable of generating deep pipelines and is not specific to this particular class of circuits.

Figure 7.1 and Figure 7.3 show the functional modules in our non-pipelined and respectively, pipelined, examples and the queues that interconnect them. In the pipelined version, the rules describe the structure of the hardware pipeline and *not* the program that executes on it.

7.3.1 Non-pipelined Specification

– Modules

```
<INC r> = im(pc) ->
    rf = rf(r->rf(r)+1), pc = pc+1;
<JRZ r l> = im(pc) and rf(r) = 0 ->
    pc = l;
<JRZ r l> = im(pc) and rf(r) != 0 ->
    pc = pc+1;
```

Figure 7.1: Non-pipelined Specification

The first rule in Figure 7.1 processes `INC` instructions. If the rule's enabling condition is true, the clause matches and binds the variable `r` to the register name argument of the `INC` instruction. The rule can then use `r` to refer to this argument. If enabled, the rule atomically executes the block in the right-hand-side of the arrow. The

update `rf = rf[r->rf[r]+1]` sets element `r` of the register file to be `rf[r]+1`. The other rules perform similar actions. To keep the example clear, the instruction set contains only an `INC` instruction, which increments the value in its single register argument, and a `JRZ` instruction, which tests the value in its register argument and, if the value is zero, jumps to the location in its location argument.

– State

Figure 7.2 presents the state and type declarations for Figure 7.1.

```
1 type reg = int(3), val = int(8), loc = int(8);
2 type ins = <INC reg> | <JRZ reg loc>;

4 var pc :  loc, im :  ins[N], rf :  val[8];
```

Figure 7.2: State Variables and Type Declarations for Example in Figure 7.1

Line 4 in Figure 7.2 presents the state declarations, which consist of the following state variables: a program counter `pc`, an instruction memory `im` and a register file `rf`. Lines 1 and 2 contain the type declarations for these variables. The type declarations include a 3 bit register name type `reg`, an 8 bit integer type `val`, an 8 bit integer type `loc` which represents the locations of instructions in the instruction memory and an instruction type `ins`. The instruction type is a tagged union type and contains only an `INC` and a `JRZ` instruction.

7.3.2 Three-stage Pipelined Specification

We repeat here Figure 2.2 from Chapter 2 . For our simple example, this is the final specification that we want to automatically generate.

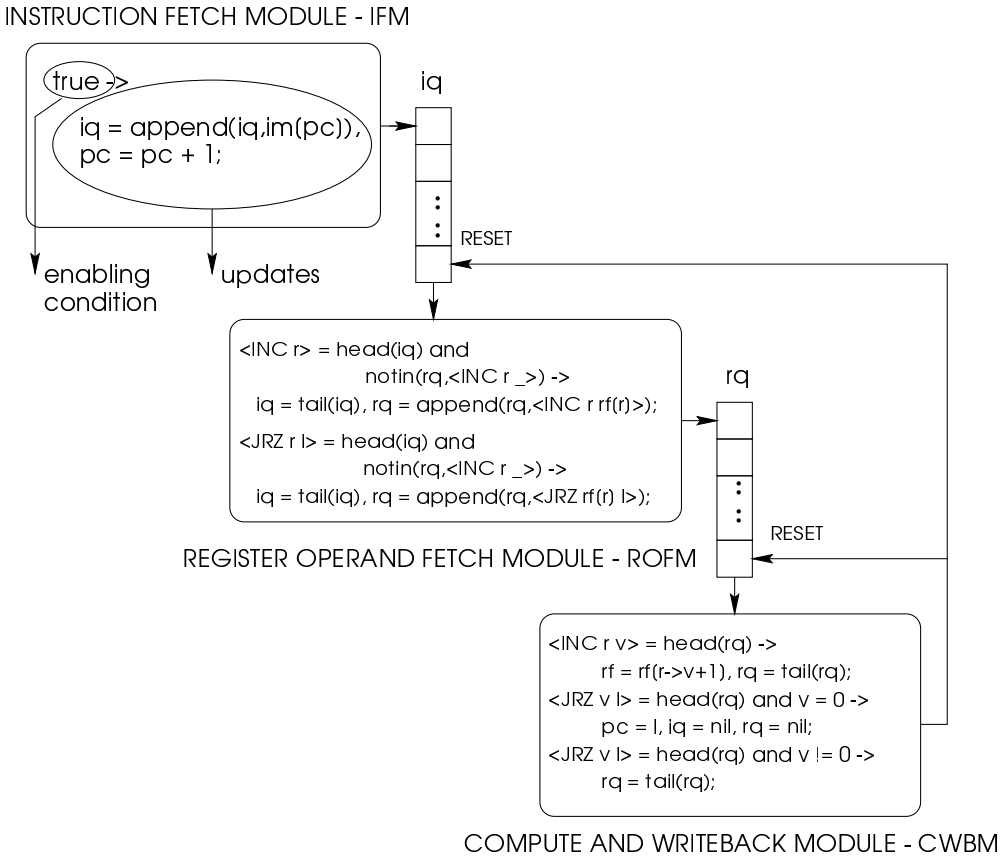

Figure 7.3: Three-stage Pipelined Specification

7.4 Pipelining Algorithm

This section presents the basic approach of our pipelining algorithm, states the property that it preserves and gives an intuition of why this is true, and presents the implementation of two techniques which improve the quality of the resulting circuit (stalling and forwarding).

7.4.1 Basic Approach

The pipelining algorithm starts with a non-pipelined or insufficiently pipelined specification and automatically generates a highly-pipelined, functionally equivalent specification. The algorithm repeatedly extracts an expression from a target module and creates a new module to compute the value of the expression at each clock cycle. It then uses a queue to transport the computed values from the new module into the target module from which the expression was extracted. The length of the queue is conceptually unbounded. Our synthesis algorithm implements all the queues in the final specification as finite hardware buffers. This algorithm operates on the resulting specification once the pipelining algorithm has finished. Our pipelining algorithm transforms the target module so that it reads the value of the expression from the queue instead of computing its value. Because this transformation splits computations across multiple clock cycles, it may reduce the clock cycle time of the circuit and increase its throughput.

In general, the extracted subcomputation may depend on values that are not available until after it must produce the new value. The compiler therefore speculates on the values that the subcomputation uses. If the speculation is incorrect, the circuit restores the values of any incorrectly updated variables and restarts the computation from the restored state. To enable the restoration, the transformed specification inserts the old values of any potentially incorrectly updated variables into the new queue. When the circuit encounters an incorrect speculation, it extracts these values from the queue and uses

them to restore any incorrectly updated variables to their correct values. The algorithm consists of seven phases for each further pipelining decision. The steps are illustrated using Figure 7.1, Figure 7.3 and Figure 7.5. Figure 7.5 is the intermediate two-stage pipelined specification derived from Figure 7.1 by applying the algorithm once.

- **Select Target Expression:** The selection of the target expression is driven by an analysis of the combinational path lengths in the circuit. The algorithm repeatedly determines the critical path, then chooses a target expression on this critical path. Inserting the computation of the target expression into a different stage of the pipeline removes the expression from the critical path, shortening its length. A more general approach could be implemented that uses a wider set of paths than the critical one(s) in deciding which expression to select as target. In addition to selecting the target expression automatically, our implemented system also allows the designer to drive the pipelining process by manually selecting the target expression. For Figure 7.1 the algorithm selects `im[pc]`; for Figure 7.5 the target expression is `rf[r]`.

- **Compute All Involved Variables:** The value of the target expression depends on the variables that it references. This set of variables is called the *set of involved variables.*

  For the specification in Figure 7.1, the set of all involved variables of `im[pc]` is {`im`,`pc`}. For Figure 7.5, the set of all involved variables of `rf[r]` is {`rf`,`head(iq)`}.

- **Speculate on New Values of Involved Variables:**

  The pipelining algorithm will move the computation of the target expression into a new module. This module will compute the value of the target expression in a clock cycle before the final values of the involved variables have been determined. The module therefore speculates on the final values of these variables, using the speculated

values to compute the value of the target expression. There are two kinds of speculation:

  * **Control:** Speculate on which rule will execute. For the involved variable `pc` in Figure 7.1 we speculate that the first rule will execute. This choice implies that the new value of `pc` is `pc+1`. For `iq` in Figure 7.5 we speculate that the first rule in the rightmost box will execute, so `iq`'s new speculated value is `tail(iq)`.
  * **Data Hazard:** Speculate on the absence of data hazards. For involved variable `rf` in Figure 7.5, we speculate that there will be no writes to `rf[r]`.

– **Generate a Queue of Values:** Generate a queue containing the sequence of values of the target expression and transform the specification to use these values. For each rule that originally read the target expression:

  * Modify the rule so that it now reads from the head of the new queue.
  * Replace all occurrences of the expression with the value read from the queue.

– **Update Involved Variables:** Augment the new module to update the involved variables with their speculated values. This operation ensures that, during the next clock cycle, the specification will generate an appropriate next value for the target expression.

  Figure 7.5 presents the results of this transformation for the specification in Figure 7.1 and target expression `im[pc]`. Figure 7.3 presents the results of this transformation for the specification in Figure 7.5 and target expression `rf[r]`.

– **Augment Queue to Handle Failed Speculations:** If the speculation is incorrect, the specification must restore the *updated variables* (the set of all variables updated as a result of the speculation)

to their correct values. The algorithm therefore augments the generated queue with the values of the updated variables before the speculation. The set of updated variables for `im[pc]` is {`pc`}. The set of updated variables for `rf[r]` is {`rf,iq`}.

The algorithm transforms the specification to detect incorrect speculations and, when necessary, use the values in the queue to restore the correct state of the system and clear the queue. The system will therefore restart from a consistent state.

For the non-pipelined example in Figure 7.1, Figure 7.4 shows the resulting specification after the algorithm executes this step.

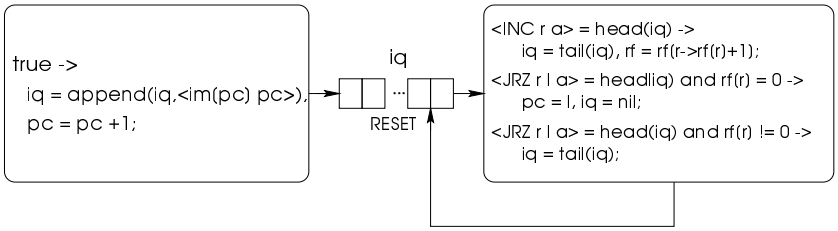

Figure 7.4: Specification After Queue Augmentation for Handling Failed Speculations

– **Remove Unused Values from Queue:** After updating all the rules that read the target expression, eliminate all the fields of queue entries that were saved and never used again.

  As we notice in Figure 7.4, field `a` of `iq` is never used, so we can safely remove it from the queue to obtain the specification in Figure 7.5.

Correctness

The pipelining transformation preserves the property that every register and memory variable observes the same sequence of values after as before pipelining.

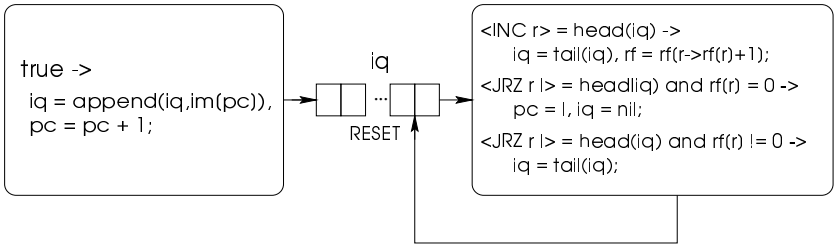

Figure 7.5: 2-stage Intermediate Specification

Informally, the pipelining transformation consists of two main steps: (1) split and replace some pipeline stage with two new stages and (2) speculate to avoid that the execution of the operations in the two new pipeline stages be serialized.

- Step (1): Contains "Select Target Expression", "Compute All Involved Variables" and "Generate a Queue of Values".

  After (1), the operations take place in the same order and generate the same values as before the split. The only difference is that producing a result takes now $n+1$ shorter clock cycles rather than n larger ones.

- Step (2): Contains "Speculate on New Values of Involved Variables", "Update Involved Variables", "Augment Queue to Handle Failed Speculations" and "Remove Unused Values from Queue".

  Once the execution has passed the pipeline stage that computes the next value of the target expression, the circuit will only use the speculated values of the involved variables. If all these speculations are correct, the circuit executes the operations in the same order as if it instead stalled until the updated values became available, just some number of cycles earlier. If at least one of the speculations is incorrect, our algorithm restores all the updated variables to the point before the speculation and clears all the queues between the point where the speculation started and the point where

it was determined to be incorrect. These actions ensure that there is no speculative operation left on fly that is using incorrect values. Effectively, they make the circuit act as if it stalled until the correct values became available, since all the incorrect, speculatively executed operations have been discarded.

7.4.2 Optimizations

We next present how our algorithm generates logic that implements two techniques — stalling and forwarding — which can improve the quality and performance of the automatically pipelined circuit.

We first present the circuit that responds to incorrect speculations by restoring saved state. Figure 7.6 shows the transformation schema for a rule R_i

$$R_i\texttt{:}\quad \texttt{e = head(str) and } P_i \rightarrow A_i$$

that reads some target expression `TE`. `e = head(str) and` P_i is the enabling condition of R_i; both clauses are optional. A missing clause reads as a true clause. A_i consists of all the updates performed by rule R_i.

In Figure 7.6, `IV` stands for the set of involved variables of `TE` and `UV` for the corresponding set of updated variables. `IV` is the disjunct union of two subsets: IV_{ctrl} — the set of involved variables on which the algorithm applies control speculation, and IV_{DH} — the set of involved variables on which it applies data hazard speculation. The speculated value of `TE` is `TE'`, and the set of speculated values for the elements of IV_{ctrl} is IV'_{ctrl}. A_i`.upd(IV)` returns the set of values that A_i updates the involved variables in `IV` to. A_i`(IV)` returns the updates in A_i that write the involved variables in `IV`. `newstr` is the newly generated queue of values for target expression `TE`. `dataHazard(E)` returns `true` if `head(newstr)` writes the expression `E` in the current clock cycle and at least one entry in `tail(newstr)` reads it.

Assume we apply this transformation to the specification in Figure 7.5, for `TE = rf[r]`. Figure 7.7 presents the `INC` instruction that this transformation

R_{i1}: e_1 = head(str) $\rightarrow$

    newstr = append(newstr,<e_1 TE[IV'_{ctrl}/IV_{ctrl}] UV>),

    IV_{ctrl} = IV'_{ctrl};

If control speculation:

If A_i.upd(IV) = IV'_{ctrl} then:

  R_{i2}: <e_2 TE_2 UV_2> = head(newstr) and

                P_i[UV_2/UV,TE_2/TE] $\rightarrow$

      newstr = tail(newstr),

      A_i[UV_2/UV,TE_2/TE] $\setminus$ A_i(IV);

Otherwise:

  R_{i2}: <e_2 TE_2 UV_2> = head(newstr) and

                P_i[UV_2/UV,TE_2/TE] $\rightarrow$

      UV = UV_2,   // restore UV

      newstr = nil,   // clear queue

      A_i;   // update

If data hazard speculation:

  R'_{i2}: <e_2 TE_2 UV_2> = head(newstr) and

                P_i[UV_2/UV,TE_2/TE] and

                no dataHazard(TE $\cup$ IV_{DH}) $\rightarrow$

      newstr = tail(newstr),

      A_i[UV_2/UV,TE_2/TE] $\setminus$ A_i(IV);

  R''_{i2}: <e_2 TE_2 UV_2> = head(newstr) and

                P_i[UV_2/UV,TE_2/TE] and

                dataHazard(TE $\cup$ IV_{DH}) $\rightarrow$

      UV = UV_2,   // restore UV

      newstr = nil,   // clear queue

      A_i;   // update

Figure 7.6: Restoration Schema

generates. The first clause of each newly derived rule reads the head of `rq` and binds `r`, `x` and `s` to the arguments of the `INC` instruction. The clause `notin(tail(rq),<_ r _ _>)` checks if there is an entry in `tail(rq)` that reads `rf[r]`. This is the check for a data hazard on `rf[r]` for the `INC` instruction. In case of a hazard — second rule in Figure 7.7 — all the updated variables, in our case `iq`, have to be restored to their old values. We would like to avoid having to store and carry the whole instruction queue `iq` along, up to the restoration point. In this case, a viable approach is stalling the pipeline stage that reads `rf[r]` until all data hazards are cleared.

```
<INC r x s> = head(rq) and notin(tail(rq),<_ r _ _>) →
    rf = rf[r->x+1], rq = tail(rq);
<INC r x s> = head(rq) and ~notin(tail(rq),<_ r _ _>) →
    iq = tail(s), rq = nil, rf = rf[r->x+1];
```

Figure 7.7: Roll-back Scheme for INC instructions

Stalling

Let `S` be the set of all target expressions on which the algorithm speculates. For `TE` $\in$ `S`, let $\{R_i\}$ be the set of rules that generate the speculated values of `TE` and let this queue be Q_1. Let $\{Q_n\}$ be the set of queues that the rules that write `TE` read from. To eliminate the need to restore state in case of a failed speculation on `TE`, the algorithm modifies the preconditions of all rules in $\{R_i\}$ to check that either 1) no item in any queue from Q_1 to $\{Q_n\}$ will generate a write to `TE` or 2) all items that update `TE` write the same known value. This approach implements the *stalling* mechanism; Figure 7.3 presents its results for our example.

Let R_{21} and R_{22} be the two resulting rules from splitting the rule in Figure 7.5 handling `INC` instructions. The check for data hazards is now handled by

R_{21}, which will not execute and read the target expression `rf[r]` until all the previous rules writing `rf[r]` did so:

```
R21:<INC r> = head(iq) and notin(rq,<INC r _>) →
        iq = tail(iq),
        rq = append(rq,<INC r rf[r]>);
```

Rule R_{22} does not anymore need to check for data hazards for the current target expression — in our example `rf[r]`. There is no speculation on fly regarding the absence of a data hazard and therefore no need to save the instruction queue in the newly generated queue of values for `rf[r]`. The derived R_{22} will have the form below:

```
R22:<INC r v> = head(rq) →
        rf = rf[r->v+1], rq = tail(rq);
```

Stalling trades the potentially higher throughput of speculative execution for a smaller circuit area. This trade-off requires a policy to decide when it is better to stall the pipeline or when it is better to speculate. The decision depends primarily on the accuracy of the predictions and the amount of state that needs to be saved for restoration purposes. Also, if all all the rules ahead in the pipeline will update `TE` with the same known value, the circuit can safely generate the next value of `TE` even if some rule will write `TE`. Therefore, a stalling check replacing a data hazard speculation waits until `TE` is hazard-free; a stalling check replacing a control speculation waits until all the previous rules in the pipeline can only update `TE` with the same known value.

Forwarding

Regardless of whether the algorithm speculates on the value of `TE` or stalls the pipeline, waiting for its correct value to become available, generating *forwarding* logic may increase the throughput of the circuit. To implement forwarding, the algorithm replaces the obsolete values of `TE` in Q_1 with their correct, updated values. Figure 7.9 shows how the technique updates a rule to implement

the bypass. `updatedTE` stands for the newly computed, correct value of `TE`. The `replace` primitive has the ability to express the function performed by the forwarding logic; that is, making the newly computed values readily available to previous pipleline stages before they update the corresponding state variables. In case a computation is stalled, forwarding can eliminate or at least reduce the amount of time spent waiting by making the correct value available as soon as it is computed. In case the circuit speculated on the new value of a state variable, forwarding can reduce the time spent recovering if the speculation is wrong by replacing its obsolete value with its correct, updated value.

```
<INC r x> = head(rq) →
    rq = replace (<INC r _>,<INC r x+1>,
         replace (<JRZ r _ _>,<JRZ r _ x+1>,
         tail(rq))),
    rf = rf[r->x+1];
```

Figure 7.8: Forward Scheme for INC instructions

Forwarding transforms the two rules handling `INC` instructions in Figure 7.7 into the single new rule in Figure 7.8. This rule produces a circuit that updates all of the entries in `rq` produced by rules that accessed `rf[r]` with the new correct value `x+1`.

Since forwarding reduces the amount of time that the circuit spends recovering from incorrect speculations or waiting for correct values to become available, it may increase the throughput of the circuit.

7.5 Conclusions

This chapter makes the following contributions:

R'_{i1}: e_1= head(str) and dataHazard(TE) $\rightarrow$

newstr = append(newstr,<e_1 updatedTE UV>),

IV_{ctrl} = IV'_{ctrl};

R''_{i1}: e_1 = head(str) and no dataHazard(TE) $\rightarrow$

newstr = append(newstr,<e_1 TE[IV'_{ctrl}/IV_{ctrl}] UV>),

IV_{ctrl} = IV'_{ctrl};

If control speculation:

If A_i.upd(IV) = IV'_{ctrl} then:

R_{i2}: <e_2 TE_2 UV_2> = head(newstr) and

P_i[UV_2/UV,TE_2/TE] $\rightarrow$

newstr = tail(newstr)[updatedTE/TE_2],

A_i[UV_2/UV,TE_2/TE] $\setminus$ A_i(IV);

Otherwise:

R_{i2}: <e_2 TE_2 UV_2> = head(newstr) and

P_i[UV_2/UV,TE_2/TE] $\rightarrow$

UV = UV_2, // restore UV

newstr = nil, // clear queue

A_i; // update

If data hazard speculation:

R'_{i2}: <e_2 TE_2 UV_2> = head(newstr) and

P_i[UV_2/UV,TE_2/TE] and

no dataHazard(IV_{DH}) $\rightarrow$

newstr = tail(newstr)[updatedTE/TE_2],

A_i[UV_2/UV,TE_2/TE] $\setminus$ A_i(IV);

R''_{i2}: <e_2 TE_2 UV_2> = head(newstr) and

P_i[UV_2/UV,TE_2/TE] and

dataHazard(IV_{DH}) $\rightarrow$

UV = UV_2, // restore UV

newstr = nil, // clear queue

A_i; // update

Figure 7.9: Forward Scheme

- **Approach:** It presents a new approach for automatically pipelining sequential circuits. This approach repeatedly extracts a computation from the critical path and moves it into a new stage. This stage uses speculation to generate a queue of values that keep the pipeline full. The approach reduces the clock cycle and increases the throughput of a circuit.
- **Algorithm:** It presents a pipelining algorithm that implements our approach. It also presents two extensions to the approach: stalling, which reduces the amount of area that would otherwise be required to respond to incorrect speculations; and forwarding, which increases throughput either by replacing values produced by incorrect speculations with correct values or by making new values available earlier to the stall logic.

Chapter 8

High-level Synthesis Systems

This section addresses related work in synthesis at higher levels of abstraction than the structural level. We do not discuss work at RTL level or lower. A comprehensive summary of early efforts can be found in [90] and [25]. We next discuss more recent work and point out the primary directions in high-level synthesis by identifying specification languages that embody each of the different approaches.

8.1 Concurrent Languages

Processes in concurrent programming languages communicate either in a synchronous or asynchronous fashion. In synchronous communication, if a signal (or message) is sent and there is no process (or module) waiting for it, the message is discarded. Otherwise, depending on the behavior of the send primitive, communication can be either non-blocking, which we will call asynchronous, or blocking, which we will call synchronizing or rendezvous. In most synchronous languages, processes are tightly coupled and deterministic, and communication is conceptually instantaneous. In asynchronous languages, processes are loosely coupled and generally non-deterministic, and communication takes time.

8.1.1 Synchronous System Design

The class of synchronous languages consists of specification languages like Esterel [16], Lustre [44], Argos [65], Signal [43], RSML [34], Statecharts [45] and hardware description languages like Cascade [15]. Lustre and Signal adopt a data-flow style and are well-adapted to steady process-control and signal processing applications which have substantial data processing needs. Esterel has an imperative style, while Statecharts, Argos and RSML are graphical; all of these are well-suited to describing control-intensive reactive behaviors. All the synchronous languages display the classical FSM behavior. The programmer thinks about a program as reacting instantaneously to external events. The synchronous model is identical to the zero-delay model of circuits, where timing delays are ignored. The only instructions that take time are those which are explicitly requested to do so. Each internal event takes place at a known time with respect to the history of external events. This feature, and the limitation to deterministic constructs (with few exceptions), results in deterministic programs from both functional and temporal points of view.

Lustre is a data-flow synchronous language whose formalism is very similar to temporal logics. Though more formal and therefore safer, inherently parallel, and promoting "code" reuse, data-flow has been traditionally considered asynchronous. Lustre proposes primitives and structures which restrict data-flow systems to only those that can be implemented as bounded memory automata-like programs. A Lustre program contains equations and assertions. Any variable and expression denotes a pair of a possibly infinite sequence of values of a given type and a clock — a sequence of times. Newly computed values can only be appended at the end of a sequence of already computed values. The basic clock is considered as setting the minimal grain of time within which a program cannot discriminate external events.

Signal is also a data-flow language, quite similar to Lustre, but which has a different semantic model. It is based on the concept of "programming by constraints": each Signal construct denotes a finite-memory relation between

sequences, and a program is the intersection of such relations. The Lustre model, on the other hand, it is based on functions over sequences and functional composition. A Signal program has a bounded memory but it can be relational and therefore it allows nondeterministic operations. Signal's execution scheme ensures that the program executes deterministically and without deadlock. Despite being widely considered a synchronous deterministic language, Signal is nondeterministic and not even a timed language [63].

Esterel is an imperative synchronous language that subsumes the model of communicating FSMs. The perfect synchrony hypothesis allows for a clean separation of functional behavior and timing constraints. The language has primitives for sequencing, parallelism, and instantaneous broadcast. The programming unit is the module. A module consists of an interface and the module body, which is composed of executable statements. Computations within modules are triggered by input events on a cyclic basis. The semantics of Esterel assumes that the computations required to produce the outputs of the modules do not consume time.

Argos, like Statecharts, is a graphical and automata-based language, different from Esterel and Lustre, which consider the state-transition paradigm not powerful enough to properly describe reactive systems. Argos is based upon the description of small systems by automata, which are represented graphically. Semantical decomposition of a behavior into smaller ones is supported; their graphical syntax allows large descriptions be cut into meaningful parts. This feature is not possible in Statecharts because arrows — representing events — are allowed between any two boxes — representing states — in the graphical representation, at any nesting level. In Statecharts, a system must be designed globally, and represented globally. On the other hand, this global approach makes Statecharts more concise than Argos in some cases. The communication model in Argos is synchronous broadcast.

RSML is a specification language for process-control systems that is based on the state machine model and specifically, on Statecharts. RSML borrows

many notions from Statecharts, leaving out the constructions that were either unnecessary or semantically too complicated for formal analysis. It then adds other features that enhance readability, simplicity and usability by application experts and developers.

Synchronous communication implies that signals (or messages) not treated in one step are ignored in the next one. While convenient in certain circumstances, to make sure that some message is not discarded, a designer using a synchronous language has to explicitly specify a mechanism that saves that message, to be processed at a later time. Synchronous languages are more suitable for designing circuits which must react in a timely manner to continuous environment stimuli, and therefore must discard many external events to maintain efficiency. Processes in synchronous languages are tightly coupled and communication is instantaneous. The disadvantage of using such an approach to specify a system is that this type of semantics forces the designer to think about the global timing when describing the system.

In our approach, different from the synchronous model, each rule takes time to execute and modules are loosely connected by asynchronous FIFO queues. The asynchronous queues implicitly ensure that each signal (or message) emitted gets processed — and in the order in which it was received. Despite the fact that asynchronous languages can reduce the designer time and effort, they are generally seen as inappropriate for implementing systems that require synchronous deterministic communication, as is the case for most digital circuits [12]. The reason is that, in asynchronous languages, processes are loosely coupled and communication takes time. A direct implementation of this model would suffer from this communication overhead and potentially compromise the efficiency of the resulting circuits. The primary challenge therefore is scheduling the operations in the circuit to maximize the throughput. Our approach is different in that it provides an asynchronous communication model at the design level, while providing a compiler able to automatically generate efficient, synchronous, pipelined implementations. The main advantage of this

approach is that it decouples the design of the different modules in the system without sacrificing the efficiency of the resulting implementation. This kind of specification encourages the reusability of the modules, and makes it easier to describe the behavior of each module because of the locality of the actions in each module. Our scheduling algorithm is a crucial vehicle to generate a synchronous, tightly coupled, pipelined implementation from a specification written in our language.

8.1.2 Protocol Specification

Another proposal from the Electronic Design Automation (EDA) community is adapting system design tools to the needs of high-level system design. The languages used by these tools, like SDL [55], Lotos [87] and Estelle [54], were originally designed for protocol specification and simulation and are all based on the extended finite state machine (EFSM) model.

SDL, standard in the telecom industry, models a system as a set of EFSMs connected via infinite FIFO queues. Each EFSM may have local variables; global variables are not allowed. Communication between SDL processes is asynchronous. An input queue is associated with each process. Signals are buffered in this queue until the process consumes them. SDL92 has introduced remote procedure calls as a step to adding synchronous communication capabilities. SDL has a complicated semantics mainly because of its complex and difficult-to-implement synchronization mechanisms.

The semantic core of Lotos is based on Milner's CCS [72]. Both CCS and Lotos, like CSP and Occam, implement sequential processes that communicate via rendezvous. None of these languages has a timed concurrent execution model and by neither our definition of synchronicity or the definition in [63] is it a synchronous concurrent model. At rendezvous points, the behavior of these languages can be nondeterministic. In Lotos, the main communication mechanism is based on binary asymmetric rendez-vous. Events must occur

simultaneously in two behaviors for communication to take place. Lotos also supports n-ary symmetric rendez-vous with choice and asynchronous communication via queues placed between EFSMs.

The communication mechanism in Estelle is implemented by asynchronous FIFO queues. The main notion in Estelle is that of a module and its instances. A module specification describes that module and a hierarchical structure of its descendant modules. Protocols are specified in terms of submodules connected by channels. Estelle specifications are nondeterministic. Riesco, Tuya and Gonzalez develop Synchronous Estelle [78], a language for specifying distributed reactive systems. Synchronous Estelle is based on standard Estelle, and contains additional constructs from Statecharts. Their idea is that the synchrony hypothesis does not hold when the reaction of the system depends on an execution distributed over a set of machines connected through communication channels. On the other hand, having to treat all the events generated by the environment — as Estelle would do — is annoying and inefficient.

Most of these languages are generally viewed as not suitable for designing hardware because of concerns regarding how the details of the implementation phase would be handled, the speed at which designs could be created and the ability to include legacy code. It is worth mentioning that, in the asynchronous model of execution, these languages have capabilities very similar to our our language. One of the important contributions of our work is that our compiler has the capability of deriving an efficient synchronous, pipelined schedule of all the operations in the system, in a timely manner. This marks a potentially significant improvement over previous approaches using languages with similar capabilities. Given the language similarities, we can envision using our technology to generate efficient hardware from these languages.

8.1.3 Other approaches

In languages like CSP [48], Occam [21] or ADA [83], processes are loosely coupled and an arbitrary amount of time can pass between announcing the desire to communicate and the completion of the communication; these languages therefore use a synchronizing form of communication — the rendezvous. A CSP program is a collection of sequential processes that execute concurrently. CSP has been successfully used as the basis for Occam, and more recently extended to address tasks in hardware codesign [79, 61, 62]. Concurrent processes in Occam communicate values through channels. Each channel provides unbuffered, unidirectional, point-to-point communication between two concurrent processes.

A program in Unity [26] — a computational model rather than a programming language — consists of a variable declaration section, the specification of the initial values for each of the variables, and a set of multiple-assignment statements. The execution of a program starts from any state that satisfies the initial conditions and repeatedly selects an assignment statement to execute. The choice is nondeterministic and such that every statement is selected infinitely often. In this sense, the model of execution in Unity and our approach are very similar and we could use our technology to generate hardware from Unity programs. The Unity model incorporates both unbounded and bounded asynchronous point-to-point communication and CSP-like rendezvous communication.

Communicating Reactive Processes [13](CRP) is a programming paradigm that aims to unify synchronous and asynchronous languages. In CRP, a set of individually reactive synchronous processes is linked by asynchronous communication channels. Technically, CRP unifies Esterel and CSP; the communication model is taken from CSP, and therefore is the rendezvous.

The rendezvous involves unbuffered communication; the sender must wait until the receiver is ready to communicate. This model (as well as the synchronous

communication model) does not enable efficient decoupled design. In asynchronous models, like ours, modules (or processes) send and receive messages at any time, provided that the input queues are not empty. Connecting the modules in the system via unbounded queues allows the circuit designer to reason about and specify each module in isolation. Decoupling the design of the different modules makes the specification process faster, easier, and less error-prone, and encourages reusability and formal verification.

8.2 Hardware Description Languages (HDL)

HDLs like VHDL [5] or Verilog [4] use a model of concurrency in which processes communicate using signals. A signal is a direct physical connection with no buffering and with dynamic synchronization overhead. At behavioral level, the functionality of a circuit is described using a language similar to a software programming language, but which includes additional abstractions necessary for specifying hardware behavior, like timing constructs and concurrency. Designed for formal verification and synthesis of communication protocols, SUAVE [6] improves the communication features of VHDL by providing bounded or unbounded message buffers.

Our specification language is not an HDL-type language. Our use of update rules that execute atomically, sequentially and asynchronously at the specification level makes it possible to think about each module in isolation, unlike for HDLs, where concurrency issues are generally and at least partially, solved by the designer rather than the system itself.

8.3 Software Languages

There are a few obvious advantages of using a software programming language to write models of both hardware/software systems and hardware circuits;

the models can be quickly evaluated, verified and directly compiled without the need to manually translate them into a synthesizable subset of HDL. The drawbacks of using programming languages for hardware design are the lack of proper abstractions regarding concurrency, structural information, circuit constraints, and timing constructs.

Different subsets of C/C++, Java and C-like description languages have been defined for synthesis. Compilogic [84] proposes a solution for translating C into RTL Verilog and assumes pointers are either memory references or parameters passed by reference without aliasing. SpC [81, 31] tries to synthesize the full ANSI C standard and addresses C programs with pointers to variables. The RAW [9] project is capable of taking correct sequential C and Fortran programs and automatically parallelize them to target an array of configurable entities. ADAS [77] is an application-driven design automation system for microprocessor design. Starting from a description in a specification language which is a subset of Standard Prolog, Fiper and Piper determine the hardware resources and the control necessary to implement the instructions. Piper performs pipeline dependency analysis and stage assignment. In general, researchers that favor specification and synthesis from software-like languages use one of the following three approaches:

- **The library extension approach:**

  This approach is based on developing class libraries for the concepts that do not exist in software languages, but are essential to capture hardware semantics. These include concurrency, constraints, structural information and timing information.

  Scenic [39] from Synopsis is a synthesizable subset of C which uses C++ constructs; it can handle any C/C++ code except for pointers and dynamic memory allocation. Scenic inherits some constructs from Esterel, which it implements as C++ classes, but, unlike Esterel, it is not truly synchronous. The semantics for concurrency is similar to that of CSP and processes communicate via signals.

Scenic is a simulation-oriented language which does not provide automated synthesis without special knowledge from the designer.

The work by Young et al. [91] at Berkeley presents a design and specification method for embedded systems that is based on Java. In their methodology, the user writes a specification in Java; this program is incrementally and semi-automatically transformed to refine it into a program that is valid with respect to given restrictions and class-library extensions. The restrictions on the usage of Java constructs are applied in order to ensure that programs are consistent with the Abstractable Synchronous Reactive (ASR) model — a model that has properties suitable for embedded system specification. ASR is based on the same synchronous approach as Esterel, Lustre and Statecharts. This method is based on library extensions, the same approach used by Ptolemy [19] and Scenic.

SystemC [3] is a C++ modeling platform that came out as the result of the initiative of leading EDA, IP, semiconductor, systems and embedded software companies to create an Open Community Licensing standard for system-level specification. The fundamental building block in SystemC is a process, which can be either synchronous or asynchronous. Processes communicate through a special type of signal called channel. Channels implicitly provide the necessary handshaking to ensure correct delivery of the data from the sender to the receiver. Channels cannot be infinite FIFOs/mailboxes.

– **The language extension approach:**

This approach extends existing software languages with additional constructs that help in describing hardware circuits.

Transmogrifier-C [38] is based on C and has additional hardware-specific constructs. Both the Transmogrifier-C and the Programmable Active Memory (PAM) [89], which uses a C++ syntax, are driven

by hardware description needs. A description is frequently not meaningful, or even illegal as a C/C++ program. Cones [85] is an automated synthesis system that takes behavioral models written in a C-based language [22] and produces gate-level implementations. The C model describes the behavior of the circuit during each clock cycle of sequential logic and does not support unbounded loops and pointers.

Reactive-C [17], SpecCharts [46] and ECL [60] are based on extending widely used software languages with constructs from synchronous languages described earlier in this chapter. Reactive-C extends C with reactive Esterel-like constructs and compiles descriptions directly to C. SpecCharts extends StateCharts with the ability to specify computations in C within each state and on transitions between states. ECL also adds explicit Esterel constructs to C; the constructs target waiting, concurrency and pre-emption. The syntax of an ECL program is C-like, with the addition of a module. A module is a subroutine that can take signal parameters. The communication model is based on the exchange of signals between and within *modules* — the only major addition to the C syntax besides the Esterel constructs. An ECL specification is compiled into a reactive part — the control portion of the system — with fully synchronous semantics, and a pure data portion that has C semantics. The control portion is equivalent to an EFSM and can be implemented either in hardware or software. The data portion in implemented in software. The language is synchronous and deterministic. With manual intervention, ECL allows a mix of synchronous and synchronizing communication. A synchronous implementation collapses the whole specification into a single EFSM for the whole system, while a synchronizing implementation simply interconnects the ECL modules as processes communicating via sig-

nals.

SpecC [37] is a superset of C and provides special language constructs for modeling concurrency, state transitions, structural and behavioral hierarchy, exception handling, timing, communication and synchronization. SpecC separates the communication and computation at higher levels of abstraction. A specification consists of a bunch of hierarchically composed *behaviors* which communicate through shared variables and/or channels. Communication channels which are useful for a specification model are those with basic synchronization (one-way and two-way handshaking) and channels for data communication (blocking and non-blocking FIFOs). Channels are mapped to buses. Scheduling of the behaviors mapped to each processing element is used to serializes the execution. Communication in the specification implies nothing about the way it will be implemented later.

Data Parallel C contains a set of extensions to Standard C that supports programming of data parallel applications. In the data parallel programming model, the designer specifies the distribution of data at high-level, and the program is written as if the data were globally addressable by all processors. There is only one logical execution thread. The compiler manages the creation of the parallel tasks and data transfer between them. Data Parallel C languages have been used in Splash 2 [41] and CLAy [40] to program arrays of FPGAs. They both are able to synthesize hardware from semantically correct programs.

– **The new language approach:**

This approach creates a new specification language that may or may not be based on one or more hardware/software languages, but whose main concepts and usage are similar to those of a software language, without making an attempt to be compatible with any

existing language.

The Olympus/Hercules system is designed to support mainly ASIC synthesis from HardwareC [57], a C-like syntax behavioral language with cycle-based semantics. HardwareC supports hardware descriptions with both procedural and declarative (e.g. interconnection of modules) semantics, and generates synchronous logic implementations. A system is a block which consists of an interconnection of logic and other blocks and processes. A process represents a functionality that executes repeatedly, concurrently with the other processes in the system. Processes can communicate and synchronize through parameter passing or message passing. The message passing mechanism implements synchronizing queues with blocking send and receive constructs. The system does not support pointers, recursion and dynamic memory allocation; the algorithms have no provisions for the synthesis of pipelined structures or multi-phase synchronous logic.

Superlog [35] is a Verilog/C based language with additional features from VHDL and other programming languages like Java. For hardware design, Superlog provides common structures such as pipelines and state machines and enhanced levels of abstraction from Verilog. The language implements threads, which allow processes to synchronize and communicate at a coarser level than at clock cycle level; on the other hand, the designer must handle the process interaction explicitly.

V++ [29] is a composable language with object-oriented flavor and syntax, built on top of the synchronous language model used by Esterel, Lustre and such. Composability is important in writing well-formed programs. V++ solves the composition problem inherent in Esterel and Lustre by preventing unrestricted communication. The communication model uses messages and is similar to VHDL. The

V++ compiler implements queues to hold messages until they are ready to be received.

OpenJ [92] is a language with layered architecture, which is targeted to embedded systems. The kernel layer is an object-oriented language derived from Java and is domain-independent, modular, and polymorphic. The domain layer contains a set of well-defined languages which capture different computation models. Domain compilers exist for discrete-event, synchronous, synchronous dataflow, and C domains. In the middle sits the open layer, which contains the set of constructs whose keywords are defined by the domain layer and where parameterized constructs are exported from the kernel layer. OpenJ defines a protocol between the kernel and the vendor domain compiler such that domain languages can be defined by supersetting and/or subsetting the kernel language. Domain languages define their own concurrency models; the kernel only provides primitives for context switch management. The main goal of OpenJ is to provide a language that can capture as many computational models as possible, yet remain simple enough to be easy to use. Except C, all the computational models captured so far are synchronous. The layered language architecture can also be found in Rapide [64].

Our specification language is neither based on an existing software language nor it has the feel of one. Connecting modules by unbounded FIFO queues is an elegant, implicit way to express concurrency and communication naturally without the need to introduce language extensions, as most of the software language-based approaches do.

8.4 Data Flow Languages

Systems like Ptolemy [19], GRAPE [59], Warp [76] at CMU, SPW [1] from Cadence or COSSAP [2] from Synopsys start from block diagram languages based on a dataflow semantics and are targeted to DSP design. The blocks in the language correspond to actors in a dataflow graph, and the connections correspond to directed edges between the actors. These edges represent communication channels, implemented as FIFO queues, and establish precedence constraints. An actor fires by removing tokens from its input edges and producing tokens on its output edges.

In Synchronous Data Flow (SDF) [14], each actor produces and consumes a fixed number of tokens, and these numbers are known at compile time. It is this constraint that "synchronous" refers to. Unlike "synchronous" languages, there is no notion of clock in the system; tokens form ordered sequences, where only the relative order is important. A static schedule for the block diagram is found that executes each *actor* in the dataflow graph at least once and does not change the net number of *tokens* queued on each edge. Synchronous dataflow and its variants are popular in DSP design due to their strong formal properties (deadlock detection, determinacy, static schedulability) and ability to model both multirate (filterbanks) and nonmultirate (IIR filters) DSP applications. The SDF model can only describe synchronous systems, in which sample rates are related by rational multiples.

The Dynamic Data Flow (DDF) model does not require that the number of tokens consumed or produced be fixed; it can vary in time or depend on the data. For this reason, DDF can describe asynchronous systems, but can only be scheduled at run time. Hybrid mixtures of SDF and DDF [20] try to combine the flexibility of DDF with the efficiency of SDF.

Our approach is like DDF in that the number of elements in the queues may vary in time and the execution of a rule is data dependent. It is unlike DDF because DDF implements a run-time scheduler, while our approach provides a

statically scheduled model, though it does contain multiplexors to choose the correct output, depending on the actual values of the state.

8.5 Asynchronous Circuitry

A number of existing systems generate as output circuits implemented as asynchronous logic. Hazard and race conditions, the inherent metastability phenomenon, and the fact that logic has been slow relative to clock speeds, are the most cited reasons against asynchronous circuitry. It is a reality that the area and performance penalty for interconnection circuitry necessary in asynchronous designs is becoming less significant than it used to be. At a closer look, metastability is not unique to asynchronous systems; any synchronous system that allows nondeterministic operations such as fair mutually exclusive memory accesses incorporates metastable circuitry [69]. There have been many approaches on designing asynchronous circuits using formal language syntax such as: trace theory, Petri nets, temporal logic, regular expressions, pass expressions, etc [88], [33], [66], [73], [11]. More recently, globally-asynchronous locally-synchronous systems [68], [28], [58] have also been considered. Meng, Brodersen and Messerschmitt [70] describe a design approach based on block diagram specifications, where blocks are either computation or interconnection blocks. The connection between computation and interconnection blocks is the request-complete signal pair. Computation blocks — combinational logic, memories, multipliers — generate completion signals to indicate valid output and request data transfers. Interconnection blocks — data transfer circuits and multiplexers — handle transfers of data and control signals. The behavior of the circuits is given in the language of the guarded command of Dijkstra [32]. From such a high-level specification, a deterministic synthesis procedure is capable of synthesizing hazard-free asynchronous interconnection circuits by strengthening the preconditions of the commands. Designing efficient combinational circuits with completion signal generation proves to be

challenging, but potentially rewarding since asynchronous circuits eliminate the requirement for a global clock, and therefore avoid the increasingly important clock skew problem. Increased modularity and reusability features, along with higher hardware utilization and shorter execution times of asynchronous circuits make them potential candidates for future technologies. Meng, Brodersen and Messerschmitt's experience in synthesizing DSPs from guarded commands indicate that, in their system, processor architecture plays an important role in specifying proper descriptions for the circuitry.

Promising results in throughput and latency of asynchronous designs have been obtained for very fine pipelines by [67]. However, pipelining their application was done manually and choosing the set of transformations for this purpose was rather an ad-hoc process. There is no strong evidence that obtaining a specification suitable for implementation is a process that can be reproduced or guided automatically. In [67], Martin et al. derive the number of pipeline stages per piece of data in transfer that is necessary to obtain optimal throughput. To obtain maximal throughput, data has to be spaced through the pipeline. Modifying the number of stages to match the optimal number can be done in a few ways, including explicitly introducing buffer stages, which increases the circuit area. We want to be able to implement our FIFO queue abstraction as the smallest number of registers such that we do not introduce deadlock while the algorithm schedules the operations for maximum throughput. A small fixed length for the FIFO queues may very well not be the desired optimal length in an asynchronous implementation.

8.6 Other Related Work

Several specification and verification systems have taken an approach similar to ours, based on describing the behavior of a system by a state transition system [26, 49]. Closely related to our research, Hoe and Arvind [50] develop a method for hardware description and synthesis based on an operation-centric

approach.

Systems based on hierarchical Production Based Specifications [80] (PBS) specify a circuit as a hierarchy of production rules. A production is recognized if its precondition — usually a composition of subproductions — is satisfied. Actions associated with each production specify the set of updates to the state of the circuit that occur when the production is recognized. The simplicity of PBS comes from the local nature of each production, which allows the designer not worry about the explicit construction of the global flow and specify additional functionality by just adding concurrent productions. The actions for a given behavior are described locally, even if possibly simultaneous actions can be described elsewhere. PBS also supports reusability of the productions. This model has similarities with our approach in that designing a circuit consists of specifying a set of components without the need to reason about the global flow explicitly. As a result, both PBS and our approach have the advantages of being modular in specifying the timing of the operations in the circuit (temporal modularity), and of supporting reusability of entire or parts of specifications. On the other hand, PBS is synchronous and does not have queues. Therefore, PBS does not provide modularity in specifying a circuit as a set of modules loosely connected by queues (spatial modularity). We believe that spatial modularity reduces the designer time and effort and makes specifications easier to understand and formally verify.

Chapter 9

Future Work

While our system is capable of generating efficient implementations for a large class of applications, there are behaviors and optimizations specific to microprocessors that the current framework does not support. In this chapter we discuss several such extensions.

9.1 Out-of-Order Speculative Execution

Out-of-order execution requires a bit more support in the language than bypassing. Concretely, we need the following:

- A primitive `notinbefore(n,q,e)` that returns true if there is not an element `e` in `q` with the instruction number less than `n`; otherwise return false. The instruction number is conceptually a counter that gets incremented every time a new instruction is fetched from the instruction memory and can be attached to the instruction at that time. This information is necessary when checking for hazards with previous instructions and flushing instructions following speculatively incorrect issued instructions.

- A primitive `concat(q1, q2)` that returns the concatenated queue, but leaves `q1` and `q2` unmodified. This primitive may be used if, for example, the designer decides to have a separate queue for stalled instructions at each stage in which hazards are checked for. Hazard checking is done for all active — speculative or not — and stalled instructions.
- A way to express the following action: "try to issue the next instruction in the queue as long as the architecture supports more concurrent instruction execution". This abstraction is also used when modeling superscalar processors. While it is feasible to have a rule for each queue element testing whether that instruction can execute or it needs to stall, having some meta-programming support for replicating rules would make the specifications more elegant. Enhancing our language to provide that support is a relatively easy job.

9.2 Exceptions in Pipelined Architectures

Exceptions are of two kinds: synchronous and asynchronous. An exception is synchronous if the exceptional event occurs at the same place every time the program is executed with the same data and memory allocation. Asynchronous exceptions are either hardware malfunctions or are caused by devices external to the processor and memory and can be usually correctly handled after the completion of the executing instruction.

- We can recognize a synchronous exception by checking for the specific exceptional condition in the corresponding rule.
- For asynchronous exceptions, we can have a last rule in the specification that, at the end of each clock cycle, checks whether any asynchronous exception was raised during that cycle.

We can model exceptions for speculative, out-of-order execution in a pipelined processor as follows. When a rule recognizes an exception, it records the exception in a reorder buffer and saves the program counter of the faulting instruction producing the exception. If a speculation is incorrect, the rule that treats this case flushes all the instructions after the speculated instruction, including all the exceptions. When an exception reaches the head of the reorder buffer, the corresponding rule executes the exception handler and restarts the instruction stream by restoring the program counter to its saved value.

9.3 Resource Constraints

Our approach is targeted towards maximizing the concurrency in the resulting circuit. One of the simplifying assumptions we made when implementing the system is that there are no resource constraints. Using a dedicated resource policy derives minimum bounds on latency for the corresponding problem if introducing resource constraints.

It is easy to incorporate resource constraint definitions into our specification language. If we do so, our minimum-latency scheduling algorithm must be replaced by a search of the solution space for the resource-constraint minimum-latency scheduling problem, for different values of the constraints. We then pick the best Pareto point[1] for area/latency optimization, for a given clock cycle. Because the resource-constraint scheduling problem is intractable, the scheduling algorithm could be based on heuristic algorithms like list or force-directed scheduling. Resource sharing and binding can be done after scheduling.

We may also want to declare the desired number of read and write ports for memories. In this case, the scheduling algorithm must stop rules from firing if

[1]The Pareto points are those points of the design space that are not dominated by others in all objectives of interest. Their image determines the trade-off curves in the design evaluation space.

the number of concurrent reads or writes exceeds the given number of ports, much in the same way it had to do that if a functional resource of some type got exhausted.

9.4 Timing Constraints

These constraints are also called latency constraints. They express the minimum/maximum timing separation between operation pairs in a circuit and data rates for pipelined systems. Relative minimum constraints between two operations can be used to ensure that an operation follows another one after at least a given number of clock cycles, regardless of the existence of a dependency between them. Likewise, maximum timing constraints limit the maximum number of cycles between two operations. By associating a unique integer number with each rule, we can specify a timing constraint between two operations by declaring a constraint between the two rules that execute those operations. In practice, scheduling under timing constraints in our system doesn't make much sense in the absence of multicycle operations. In the presence of multicycle operations, the existence of a schedule under timing constraints cannot be guaranteed if there exists at least one maximum timing constraint. This is true for the following reasons:

- An upper time bound between the starting time of two operations can be insufficient for the first operation (and possibly other intermediate operations) to finish execution.
- Minimum timing constraints may conflict with maximum timing constraints.
- It is impossible to determine statically whether some rule will be ready to execute within a given time frame because the truth values of the enabling conditions of the rules are generally data dependent.

Minimum timing constraints can cause the circuit to go idle until the timing constraints are satisfied if the only rules that are enabled do not satisfy these constraints.

9.5 Implementation Constraints: Cycle-Time Bounds

As presented, our synthesis approach gives the designer no explicit control over the final value of the clock cycle time. The developer writes the specification and from now on, he does not have any control over the resulting implementation; the compiler automatically generates an implementation of the circuit that is optimized for concurrency. Sometimes, this policy can lead to circuits with longer clock cycles than the developer may have had in mind when he developed the specification. This section discusses how to change our synthesis system to generate the extra control necessary to enforce a maximum bound on the clock cycle time. Chapter 7 presents a complementary method for shortening the cycle time of a circuit. There, the pipelining algorithm can repeatedly remove the computation of an expression from the current critical path, until the length of the critical path is less or equal to the target clock cycle time.

Our framework does not currently allow the designer to specify and synthesize circuits implementing multicycle operations. Each rule is atomic and therefore its execution has to complete within the duration of a clock cycle. The execution model repeatedly chooses an enabled rule and executes it. At each clock cycle, each rule is evaluated and gets a chance to execute. If multiple rules simultaneously evaluate their preconditions to true, they are all going to execute in the current clock cycle, either in parallel if there are no data dependencies or sequentially if there are. This implementation policy is aimed towards deriving circuits whose clock cycle time best enables exploiting the

concurrency of the system.

Our policy is at one extreme end of the *concurrency/clock cycle* trade-off. We can easily implement another policy that derives a schedule with chaining[2] for a given value of the clock cycle. The main difference from the current policy is that a rule will now test an additional enabling condition before being allowed to execute. This condition — let's call it "it chains" — tests whether the overall propagation delay of the current rule and of all the rules that already executed in the current cycle does not exceed the given cycle time. Since at synthesis time we don't know whether a rule is going to execute at some clock cycle, its propagation delay and the overall propagation delay in the circuit can take more than one value. The overall delay value depends on the propagation delays of the circuitry computing (1) the truth value of the current rule's precondition, (2) the new values of the state variables that the current rule updates, (3) the propagation delays of previous rules and whether these rules had executed in the current cycle or not.

To give a intuition of why (3) is true, notice that computations of the precondition and the expressions of a rule may refer to different instances of the same state variable. Taking the initial value of a variable at the beginning of the clock cycle does not incur any additional propagation delay other than that of a read operation; taking the value of the variable produced by a previous rule means waiting for that value to be first produced. The first operation can execute in parallel with the rest of the system, while the second one must execute in sequence, after rule `R` producing the intermediate value has finished execution. In turn, the overall propagation delay after rule `R` can have different values depending on a number of factors, some of which being (1) the delays for computing the precondition and the updates of `R` and (2) whether `R` actually executed or not.

[2]Scheduling with chaining uses the propagation delays of the combinational resources instead of the usual integer execution delays. Two or more combinational operations in a sequence can be chained in the same execution cycle if their overall propagation delay does not exceed the cycle time.

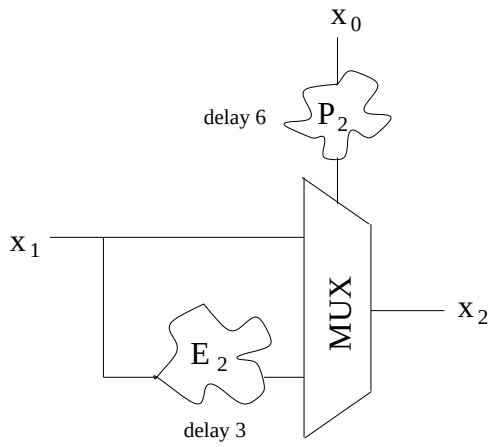

Figure 9.1: Computing Version 2 of State Variable x

To illustrate these cases, Figure 9.1 shows the two possible values of state variable $\mathtt{x}$ after rule $\mathtt{R}_2$: the previous value $\mathtt{x}_1$, and the newly computed value as $\mathtt{E}_2(\mathtt{x}_1)$. The truth value of $\mathtt{P}_2(\mathtt{x}_0)$ selects the appropriate MUX entry. For the sake of simplicity, let's assume the MUX does not have any propagation delay of its own and delays are represented as integer numbers. The integers in the figure represent the propagation delays of the adjacent circuitry (represented as irregular clouds in the picture), once the state variables that they use are available. Therefore, computing $\mathtt{P}_2$ takes 6 units once $\mathtt{x}_0$ is available and $\mathtt{E}_2$ takes 3 units once $\mathtt{x}_1$ is available. $minPropagDelay(\mathtt{R})$ and $maxPropagDelay(\mathtt{R})$ stand for the smallest and the largest overall propagation delays in the circuit after rule $\mathtt{R}$ was evaluated. Let's assume for the clarity of the explanation that in our case these values depend on whether rule $\mathtt{R}$ actually executed in the current clock cycle or not.

- The propagation delay of $\mathtt{R}_2$ will always be 6 if the value of the MUX select takes more time to compute than the computation of the two alternatives. It depends only on the propagation delay of rule $\mathtt{R}_1$ whether the propagation delay after rule $\mathtt{R}_2$ can take more than one possible value. More accurately, if $3 + maxPropagDelay(\mathtt{R}_1) <= 6$

then the overall delay for rules $\mathtt{R}_1$ and $\mathtt{R}_2$ is always going to be 6, regardless of whether either of the two rules actually executes or not. In our case, propagation delays for rule $\mathtt{R}_1$ of minimum 1 and maximum 2 would fit the scenario. If $3 + maxPropagDelay(\mathtt{R}_1) <= 6$ is not true, the overall propagation delay after rule $\mathtt{R}_2$ can take more than one value. This scenario shows a case where whether rule $\mathtt{R}_1$ executes or not is not relevant to determine the overall propagation delays in the circuit, but the propagation delays of both rules $\mathtt{R}_1$ and $\mathtt{R}_2$ are.

– If $3 + minPropagDelay(\mathtt{R}_1) < 6 < 3 + maxPropagDelay(\mathtt{R}_1)$, then if rule $\mathtt{R}_1$ executes, the overall delay after rule $\mathtt{R}_2$ is 6 if the precondition of $\mathtt{R}_2$ is false, otherwise it is $3 + maxPropagDelay(\mathtt{R}_1)$. Values of 2 and 4 for minimum/maximum propagation delays of $\mathtt{R}_1$ fit the scenario. This shows a case where it is crucial whether rule $\mathtt{R}_1$ executed to compute the overall propagation delay in the circuit.

For given propagation delays of the individual hardware components of a circuit and for a given clock cycle value, more or fewer rules can be scheduled for concurrent execution depending on the specific values of the state variables at a certain time. This is true because it is the values of the variables which determine whether the precondition of a rule is true or not in the current clock cycle and this influences the value of the overall propagation delay of the system.

9.6 Large Memories

The current implementation of our system uses synchronous memory arrays to implement any state variable of type *arrayType*. For efficiency reasons, we may want to implement large memories using RAMs (Random Access Memory) instead. This operation involves a straightforward modification in the

phase that generates synthesizable Verilog. RAM-type memory elements are supported in most synthesis packages — including Synopsys and Cadence — through instantiation in the Verilog source code. The types of RAM supported are usually single- and dual-port synchronous RAM and sigle-port level-sensitive RAM. We can therefore make use of the RAM implementations already in the library packages. Although RAM can be also described behaviorally, the standard synthesis tools currently synthesize it to inefficient latch- or register-based implementations.

9.7 Register Renaming

Register renaming is the technique used to eliminate name dependences so as to avoid Write After Read (WAR) and Write After Write (WAW) hazards between state variables stored in register operands. A name dependence occurs when two instructions of an instruction-set architecture use the same register or memory location, called a name, but there is no flow of data between the instructions associated with that name. In our system, register renaming avoids invalidating the stream of speculatively generated register values whenever the register is written, but the system speculated otherwise. To implement register renaming, we need a primitive `new`, which creates a new name for a value to be produced in the future. Each register file entry will contain either a concrete value or a name produced by `new`. This name is of a virtual register in an extended set of virtual registers created for renaming purposes. We use the `replace` primitive to update all name fields of entries in a given queue with their new, correct value. Performing this operation avoids restoration when data hazard speculation goes wrong such that it would otherwise create a WAR or WAW hazard.

Chapter 10

Conclusions

The goal of this thesis was to provide a new approach for hardware synthesis that is at the same time easy, concise and efficient. The designer uses a design language based on connecting modules with asynchronous queues. The synthesis algorithm eliminates the potential inefficiency associated with a direct asynchronous implementation by automatically generating a coordinated global schedule for all operations in the system. This schedule is used to generate an efficient and fully pipelined synchronous implementation.

The primary advantages of this approach include good support for concurrency, modularity, debugging, and reuse in the design language. The use of update rules provides support for formal verification and concurrency, and enables concise, behavioral descriptions. This gives the resulting implementation a better chance to correctly reflect the designer's intent. The synthesis algorithm is the key to enabling the designer to use a convenient design language while obtaining an efficient hardware implementation of the design. The global scheduling and relaxation algorithms maximize the throughput. Relaxation also reduces the clock cycle time by parallelizing the evaluation of the enabling conditions of the rules. Global scheduling eliminates the need for handshaking hardware, while applying optimizations at a global level optimizes the combinational logic.

We implemented the compiler that takes specifications in our description language and generates circuits at RTL level. We set two goals for our system: easy design of circuits using the language and high performance of the resulting implementations. To evaluate our system, we first developed a set of benchmarks written in our specification language. We then gathered a set of figures that give a qualitative measure of how well the system met its first goal: time to write the circuit specification, specification size and compiler running time to generate the corresponding Verilog implementation. We then ran the benchmarks through our synthesizer and got clock cycle and circuit area numbers for the resulting implementations. Our results provide encouraging evidence that the approach can deliver efficient implementations from concise, simple, high-level specifications.

Our approach is well-suited to systems that are naturally described as a composition of interacting sub-systems. The class of pipelined circuits is one such system, as FIFO queues are a natural way to isolate pipe stages. Nevertheless, the current implementation of our system does not provide a mechanism for specifying timing constraints. Timing constraints can model concurrency. Therefore, the lack of an explicitly parallel abstract execution model does not impose further limitations besides the ones that spring from the lack of a built-in constraint mechanism. This limitation may nevertheless make our specification language in its current form unsuitable for describing some highly optimized systems, but still allow it to be a very useful tool in experimenting with different prototypes of these circuits. Interfacing with a component which implements a timing constraint mechanism would enable efficient synthesis for a larger class of highly optimized circuits. The strengths of our compiler allows it to tackle successfully DSP and embedded systems, as well.

Bibliography

[1] wwww.cadence.com.

[2] wwww.synopsys.com.

[3] wwww.systemc.org.

[4] wwww.verilog.com.

[5] wwww.vhdl.org.

[6] P. Ashenden, R. Esser, and P. Wilsey. Communication and synchronization using bounded channels in SUAVE. 1999.

[7] P.J. Ashenden. *The Designer's Guide to VHDL.* Morgan Kaufmann Publishers, 1995.

[8] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[9] J. Babb, M. Rinard, A. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe. Parallelizing applications into silicon. In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'99)*, Napa Valley, CA, 1999.

[10] M. Ballantyne, W.W. Bledsoe, J. Doyle, R.C. Moore, R. Pattis, and S.J. Rosenschein. Automatic deduction. Technical Report STAN-CS-82-937, Dept. of Computer Science, Stanford Univ., Stanford, Calif., October 1982.

[11] T.S. Balraj and M.J. Foster. *Miss Manners: A specialized silicon compiler for synchronizers.* Advanced Research in VLSI, C.E. Leiserson, Ed. MIT Press, April 1986.

[12] G. Berry. *Real-time Programming: General Purpose or Special-purpose Languages.* G. Ritter, editor, Information Processing 89, 1989.

[13] G. Berry, S. Ramesh, and R.K. Shyamasundar. Communicating reactive processes. In *Proceedings of the 20th ACM Conference on Principles of Programming Languages (POPL)*, Charleston, Virginia, 1993.

[14] S. Bhattacharyya, P. Murthy, and E. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing*, 21:151–166, 1999.

[15] D. Borrione and C. Le Faou. *Overview of the CASCADE multi-level hardware description language and its mixed model simulation mechanisms.* Computer Hardware Description Languages and Their Applications, 1985.

[16] F. Boussinot and R. de Simone. The ESTEREL language. In *Proceedings of the IEEE*, pages 79(9):1293–1304, September 1991.

[17] F. Boussinot, G. Doumenc, and J.-B. Stefani. Reactive objects. *Annales des Telecommunications*, 51(9-10):459–473, 1996.

[18] M. Breternitz and J. P. Shen. Architecture synthesis of high-performance application-specific processors. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1990.

[19] J. Buch, S. Ha, E.A. Lee, and D.G. Messerschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, 1995.

[20] J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt. Multirate signal processing in Ptolemy. July 1991.

[21] A. Burns. *Programming in Occam 2.* Addison-Wesley, Reading, Mass., 1988.

[22] C.T. Bye, M.R. Lightner, and D.L. Ravenscroft. A functional modeling and simulation environment based on ESIM and c. In *Proceedings of the 1984 ICCAD*, 1984.

[23] F. Rose C. E. Leiserson and J.B. Saxe. Optimizing synchronous circuitry by retiming. In *Proceedings of the 3rd Caltech Conference on VLSI*, 1983.

[24] Y.-C. Hsu C.-T. Hwang and Y.-L. Lin. Scheduling for functional pipelining and loop winding. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, San Francisco, 1991.

[25] R. Camposano and W. Wolf. *High-Level VLSI Synthesis.* Kluwer Academic Publishers, 1991.

[26] K. M. Chandy and J. Misra. *Parallel program design: a foundation.* Addison-Wesley, Reading, Mass., 1988.

[27] F. Chang and A. Hu. Fast specification of cycle-accurate processor models. *Submitted to ICCD 2001*, 2001.

[28] D.M. Chapiro. *Globally-asynchronous locally-synchronous systems.* PhD thesis, Stanford University, October 1986.

[29] S. Cheng, P.C. McGeer, M. Meyer, T. Truman, A. Sangiovanni-Vincentelli, and P. Scaglia. The V++ system design language. In *Proceedings of DATE'98*, 1999.

[30] R. J. Cloutier and D. E. Thomas. Synthesis of pipelined instruction set processors. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, 1993.

[31] G. de Micheli. Hardware synthesis from c/c++ models. In *Proceedings of DATE'99*, 1999.

[32] E.W. Dijkstra. *Guarded commands, nondeterminacy, and formal derivation of programs.* Commun. Ass. Comput. Mach., vol. 18, no. 8, Aug, 1975.

[33] J.C. Ebergen. *A formal approach to designing delay-insensitive circuits.* Computer Science Notes, Eindhoven University of Technology, October 1988.

[34] N.G. Leveson et al. Requirements specification for process control systems. *IEEE Transactions on Software Engineering*, 20(9), 1994.

[35] P.L. Flake, S.J. Davidmann, and D.J. Kelf. Superlog - Evolving Verilog and C for system-on-chip design. In *International HDL Conference*, San Jose, CA, 2000.

[36] H. De Man G. Goossens, J. Rabaey and J. Vandewalle. An efficient microcode-compiler for custom DSP-processors. *IEEE Transactions on Computer-Aided Design*, 9:925–937, 1990.

[37] D. Gajski, J. Zhu, R. Doemer, A. Gerstlauer, and S. Zhao. The SpecC methodology. In *Technical Report ICS-99-56*, December 1999.

[38] D. Galloway. The Transmogrifier-C hardware description language and compiler for FPGAs. In *IEEE Workshop on FPGAs for Custom Computing Machines*, Napa Valley, 1995.

[39] A. Ghosh, J. Kunkel, and S. Liao. Hardware synthesis from C/C++. 1999.

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

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

[42] G. Goossens, J. Vandewalle, and H. DeMan. Loop optimization in register-transfer scheduling for DSP-systems. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, 1989.

[43] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming real time applications with Signal. In *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, September 1991.

[44] N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous dataflow programming language Lustre. In *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, September 1991.

[45] D. Harel. Statecharts: a visual approach to complex systems. In *Science of Computer Programming*, pages 8:231–274, 1987.

[46] D. Har'el, H. Lachover, A. Naamad, and A. Pnueli et al. STATEMATE: a working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4), 1990.

[47] S. Hassoun and C. Ebeling. Architectural retiming: Pipelining latency-constrained circuts. In *Proceedings of the 33rd ACM/IEEE Design Automation Conference*, Las Vegas, 1996.

[48] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[49] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, N.J., 1985.

[50] J. Hoe and Arvind. Hardware synthesis from term rewriting systems. In *VLSI: Systems on a chip*, Lisbon, Portugal, December 1999.

[51] U. Holtmann and R. Ernst. Combining MBP-speculative computation and loop pipelining in high-level synthesis. In *ED & TC*, 1995.

[52] I.-J. Huang and A. M. Despain. High level synthesis of pipelined instruction set processors and back-end compilers. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, 1992.

[53] P. Hudak, S. Peyton-Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. Guzman, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain,

and J. Peterson. Report on the programming language Haskell: a non-strict, purely functional language (version 1.2). *SIGPLAN Notices*, 27(5), May 1992.

[54] ISO. *Estelle: A Formal Description Technique Based on an Extended State Transition Model*, 1978. Draft International Standard 9074.

[55] ed. K.J. Turner. *Using Formal Description Techniques - An Introduction to Estelle, Lotos and SDL.* John Wiley Sons, Chap. 4 5, 1993.

[56] D. Kroening and W.J. Paul. Automated pipeline design. In *Proceedings of the 38th ACM/IEEE Design Automation Conference*, Las Vegas, 2001.

[57] D. Ku and G. De Micheli. HardwareC: a language for hardware design. Technical Report SCSL/CSL/TR-90-419, August 1990.

[58] S.Y. Kung, K.S. Arun, J. Gal-Ezer, and D.V. BhaskarRao. Wavefront array processor: Language, architecture, and applications. *IEEE Transactions on Computers*, C-31:1054, November 1982.

[59] R. Lauwereins, P. Wauters, M. Ade, and J.A. Peperstraete. Geometric parallelism and cyclo-static data flow in GRAPE-II. In *Proceedings of IEEE Workshop on Rapid System Prototyping*, Grenoble, France, 1994.

[60] L. Lavagno and E. Sentovich. ECL: A specification environment for system-level design. In *Proceedings of the 36th ACM/IEEE Design Automation Conference*, New Orleans, 1999.

[61] A. Lawrence. Extending CSP. In *WOTUG21 - Architectures, Languages and Patterns for Parallel and Distributed Applications*, UK, 1998.

[62] A. Lawrence. HCSP: Extending CSP for codesign and shared memory. In *WOTUG21 - Architectures, Languages and Patterns for Parallel and Distributed Applications*, UK, 1998.

[63] E.A. Lee and A. Sangiovanni-Vincentelli. Comparing models of computation. In *Proceedings of the 1996 ICCAD*, 1996.

[64] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):336-355, 1995.

[65] F. Maraninchi. The Argos language: Graphical representation of automata and description of reactive systems. In *Proceedings of the IEEE Workshop on Visual Languages*, Kobe, Japan, October 1991.

[66] A.J. Martin. *Compiling communicating processes into delay-insensitive VLSI circuits.* Distributed Computing, vol. 1, 1986.

[67] A.J. Martin, A. Lines, R. Manohar, M. Nystroem, P. Penzes, R. Southworth, U. Cummings, and T.K. Lee. The design of an asynchronous MIPS R3000 microprocessor. In *Proceedings of the 17th Conference on Advanced Research in VLSI, IEEE Computer Society Press, 164-181*, 1997.

[68] C. Mead and L. Conway. *Introduction to VLSI systems.* ADDISON, Chap. 7, 1980.

[69] T. Meng, R. Brodersen, and D.G. Messerschmitt. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Transactions on Computer-Aided Design*, 8, November 1989.

[70] T. Meng, R. Brodersen, and D.G. Messerschmitt. Asynchronous design for programmable digital signal processors. *IEEE Transactions on Signal Processing*, 39, No. 4, April 1991.

[71] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* Cambridge, MA, 1990.

[72] Robin Milner. *A Calculus of Communicating Systems.* Springer-Verlag, Vol. 92, LNCS, 1980.

[73] D. Misunas. *Petri nets and speed independent design.* Commun. Ass. Comput. Mach., vol. 16, no. 8, Aug, 1973.

[74] N. Park and A. C. Parker. Sehwa: A software package for synthesis of pipelines from behavioral specifications. *IEEE Transactions on Computer-Aided Design*, 7:356–370, 1988.

[75] P. B. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Transactions on Computer-Aided Design*, 8:661–679, 1989.

[76] H. Printz. *Automatic mapping of large signal processing systems to a parallel machine.* PhD thesis, Carnegie Mellon University, 1991.

[77] I. Pyo, C. Su, I. Huang, K. Pan, Y. Koh, C. Tsui, H. Chen, G. Cheng, S. Liu, S. Wu, and A.M. Despain. Application-driven design automation for microprocessor design. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, 1992.

[78] M. Riesco, J. Tuya, and O. Gonzalez. Synchronous Estelle: A language to specify distributed control systems.

[79] S. Schneider. *Concurrent and Real-time Systems: the CSP Approach.* John Wiley Sons.

[80] Andrew Seawright and Forrest Brewer. Synthesis from Production-Based Specifications. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, 1992.

[81] L. Semeria and G. de Micheli. Spc: Synthesis of pointers in c, application of pointer analysis to the behavioral synthesis from c. In *Proceedings of ICCAD'98*, San Jose, 1998.

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

[83] S. Smolka and P. Wegner. Concurrent programming primitives in CSP and ADA. November 1981.

[84] D. Soderman and Y. Panchul. Implementing c designs in hardware: A full-featured ANSI c to RTL verilog compiler in action. `http://www.compilogic.com`.

[85] C. Stoud, R. Munoz, and D. Pierce. Behavioral model synthesis with cones. *IEEE Design Test of Computers*, 5:22–30, 1988.

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

[87] K.J. Turner and M. van Sinderen. *Lotos specification style for OSI, The LOTOSPHRE Project.* KLUWER, London, UK, 1995.

[88] J. van de Snepscheut. *Trace theory and VLSI design.* Lecture Notes on Computer Science 200, Springer, 1985.

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

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

[91] J.S. Young, J. MacDonald, M. Shilman, A. Tabbara, P. Hilfinger, and A.R. Newton. Design and specification of embedded systems in Java using succesive, formal refinement. In *Proceedings of the 35th ACM/IEEE Design Automation Conference*, 1998.

[92] J. Zu and D. Gajski. OpenJ: An extensible system level design language. In *Proceedings of DATE'99*, 1999.