HIGH-LEVEL DIGITAL INTERFACES WITH LOW OVERHEAD

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Kyle R. Kelley
August 2012

This dissertation is online at: http://purl.stanford.edu/gc931zj7563

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christoforos Kozyrakis

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Oyekunle Olukotun

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

The majority of today's digital designs are coded in hardware description languages (HDLs) such as Verilog, VHDL, BlueSpec, SystemC, etc. HDLs provide useful abstractions to facilitate the design of complex systems, and although they offer diverse syntaxes for expressing hardware, they actually share similar module interface semantics. These interfaces rely on hardwired, timing-dependent communication protocols, and offer poor design-time parameterization of internal control logic, both of which impede complex system design.

In this thesis, we describe a high-level interface abstraction that improves upon the hardwired interfaces common to popular HDLs. These high-level interfaces create logically asynchronous connections between modules, allowing module timings to vary without breaking system functionality. This has a number of design advantages, including better design exploration and easier module reuse. Moreover, high-level interfaces abstract hardwired control logic as per-instance module elaboration parameters[19], further enabling module reuse.

These generic, flexible interfaces are rarely used today because they lead to timing and area overheads compared to hardwired, customized designs. To address this, we present a reachability analysis framework that can be used to identify and remove overhead from high-level interfaces in gate-level netlists[20]. This makes the synthesis results of high-level interfaces comparable to typical hardwired approaches. We use various examples from the Stanford Smart Memories project [14][31] to demonstrate the use of high-level interfaces, and how they can be synthesized into efficient implementations.

By building modules with high-level interfaces, system designers can both modify

existing designs (e.g., pipeline long paths) and reuse modules to compose new working systems, without worrying about the timing of interface handshakes. Furthermore, reachability analysis ensures high-level interfaces do not add any logic overhead compared to a hardwired interface. Therefore, we believe high-level interfaces are a useful abstraction for extending HDLs as design complexities continue growing into the future.

Acknowledgments

First and foremost, I want to thank my advisor, Mark Horowitz, for putting up with me all of these years (and even letting me return after a nearly 2 year leave). It's easy to get lost in low-level details of technical work, but Mark has an uncanny (and relentless) ability to always push for the bigger picture. While none of his students are ever thrilled to hear his favorite phrases "why don't you pop up a level?" and "what are you *really* trying to say?", I am glad that they will be forever ingrained in my mind.

I would also like to thank my committee members Christos Kozyrakis and Kunle Olukotun, for graciously agreeing to serve on my committee, and for putting up with a barage of my frantic emails in recent weeks while trying to wrap my thesis up.

Steve Richardson has been immensely helpful over the years, always eager and available to bounce ideas off of, proof-read new papers, and even helping me procrastinate by engaging in politically-charged email debates.

I've had the pleasure of working alongside many smart and talented students in my time at Stanford. My groupmates Ofer Shacham, Megan Wachs, and Zain Asgar were always immensely helpful, and certainly made time in the office more enjoyable. A special thanks goes to James Mao for feeding my addiction to distracting side projects, and being willing to argue about anything and everything.

David Harris, my undergraduate advisor at Harvey Mudd College, deserves a special thanks for introducing me to academic research, and encouraging me to continue on to graduate school.

Last, but not least, I want to thank my parents, Jill and Bob Kelley. I haven't thanked them enough for all they did for me growing up, and certainly can't do it

justice here. When I was very young, they told me that it's possible to do anything in life, as long as I put my mind to it. Fortunately (and, occassionally, unfortunately) I actually believed them, and to this day still take that advice to heart. I still have trouble explaining to them what a Ph.D. is (graduate school isn't exactly on the map where we're from), and I know during the time I've been at Stanford my dad has continued wondering why I graduated from college years ago yet still lacked a real job. Well, Dad, now I can finally say, "soon!".

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Digital VLSI systems are ubiquitous in our daily lives. Microprocessors have moved beyond their traditional roles in personal computers and high-end servers, now playing integral parts in the operation of a vast array of consumer products, ranging from automobiles to mobile phones. This growth has been primarily facilitated by the now popular Moore's Law, which for decades has continued delivering twice as many transistors per chip every 18-24 months[24]. Since each technology generation brings more processing capability (for the same energy), consumers now expect new and interesting applications of embedded VLSI technology to continue appearing on the market. Cellular phones, once relatively simple analog transceivers, have now become complex multi-core digital computers with embedded graphics and advanced image processing capabilities. One can only imagine what functionality and applications the next generation of technology products will bring.

Although Moore's Law has been a blessing for consumers, in many ways it has become a bane for designers. The exponential growth in transistors per chip has given rise to exponentially growing design complexity, as modern chips now consist of over 2 billion transistors[35][27]. This sheer number of transistors makes design difficult, and requires numerous levels of abstraction. Worse yet, this complexity makes it difficult to understand and reason about designs, creating a verification nightmare. These difficulties are reflected in the design and validation costs, where it is now estimated that well over 80% of total ASIC cost is devoted to system design and verification

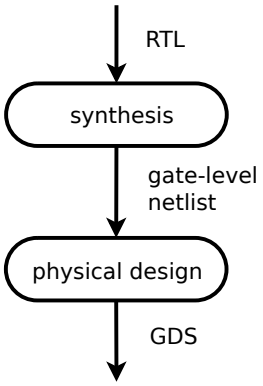


Figure 1.1: A simplified depiction of RTL design flow. RTL is first synthesized into a netlist of standard cell gates, which are then processed by physical design tools to create a full chip description for manufacturing (GDS).

efforts[16].

These complexity issues are not new; historically, the integrated circuit (IC) industry has tackled them by continually moving to higher design abstractions, which facilitate increased design complexity by hiding and/or re-using lower-level components. Originally, circuit designers drew custom schematics and laid out transistors by hand. Eventually, the common logic functions were encapsulated in standard cell libraries, allowing designers to reason about individual logic gates in their schematics rather than underlying transistors. This gave rise to hardware description languages (HDLs) and register transfer level (RTL) abstractions, which describe designs in logical code rather than schematics of gates and connections. Logic synthesis tools were created to automatically convert the RTL descriptions into gate-level netlists of standard cells, once again freeing designers of lower-level details. Similarly, placement and routing tools were created to convert these gate-level netlists into physical descriptions of transistors and wires. Figure 1.1 depicts this modern design flow. For context note that this work only focuses on improving RTL and synthesis.

Despite these advantages, the semantics of modern HDL interfaces lack flexibility that would greatly facilitate the design and reuse of system components. Section 1.1 describes these shortcomings in more detail. Section 1.2 then introduces high-level interfaces, a new proposed abstraction to overcome these limitations, and Section 1.3

discusses related work that has been done in this area. Chapter 2 shows how to build high-level interfaces with modern languages, and Chapter 3 discusses an optimization technique to ensure they remain overhead-free. Chapter 4 demonstrates the use of high-level interface abstractions on real designs taken from a chip-multiprocessor, and Chapter 5 offers some concluding thoughts and possible future research directions.

1.1 HDL Interface Limitations

A common approach to building complex systems in many fields of engineering involves partitioning the functionality into encapsulated design components (or modules). This modularization generally makes design easier since the components can be built by different people, and also reused in various places. To maximize reuse, it is important that modules be built with generic interfaces to allow them to work in new environments.

In this section we highlight some shortcomings of typical HDL interfaces, motivated purely by hardware design needs. Since this is a common engineering issue, we also examine how software languages have approached it, gaining intuition about interface engineering and motivating possible solutions to make them more flexible in hardware.

The two most common HDLs that implement the register transfer level (RTL) abstraction are Verilog and VHDL. Although these languages have different syntaxes, they have similar expressivities and without loss of generality suffer from similar shortcomings because they're both just RTL descriptions. Hence, we will focus on Verilog in our discussion of RTL.

1.1.1 Pain Points

Intermodule Communication

The lack of a flexible and robust communication abstraction in modern HDLs ultimately makes it difficult to both (1) alter an existing functional system, and (2) reuse existing modules to compose a new functioning system. Indeed, designers will

agree that most of the difficulty in building a large system is not in building/verifying the individual components, but in getting the components to communicate properly. This is particularly problematic when modules are built by different people, since this requires making assumptions about the other's behavior.

The conventional approach requires designers to agree on the behavioral specification of interface signals, which describes both their ordering (i.e., in what order are messages sent/received) *and* their logical timings (i.e., number of cycles for a request/response). The author of each module then designs them to operate according to this specification, and they are able to communicate harmoniously.

We argue that this approach is too strict because of its dependence on strict logical timings. For example, it is common in later design stages to close timing by adding pipeline registers along critical paths. Since this alters a module's latency, it is likely to break communication functionality at its interface with neighboring modules that were designed for the original logical timing. Similarly, it is common to have various different implementations of a particular module to explore different architectures and topologies, allowing wider flexibility in the energy-performance space. Again, the lack of timing flexibility in RTL communication makes modules with different timings incompatible with the original system. This ultimately limits and inhibits the design-space exploration that is possible with conventional HDLs.

Control Logic

Since RTL does not have any specific notion of control logic, it is typically described using a canonical finite-state-machine (FSM) style, where one RTL module encapsulates the entire FSM. A register element holds the state, and combinational logic describes the next-state and output functions. This style is typically easily readable by others, and can be recognized by synthesis tools to perform FSM-specific optimizations. However, this hardcoded style lacks parameterization. The typical approach to reuse and tweak control logic for different environments would be to fork the code, which creates maintainability issues. Instead, we want control logic that is parameterized and abstracted in interfaces, allowing the behavior of module instances to be more easily varied.

As we continue to build more advanced systems with HDLs, it will become more common to desire changes in control logic to properly tune the design for the target application. As an example, a flexible multiprocessing system might need to operate with different cache-coherence protocols. While the underlying architecture remains the same, the memory controller should perform slightly different operations depending on the specific coherence protocol being used. It would be far easier to perform these modifications if control logic were parameterized in module interfaces, rather than having forked hard-coded implementations.

1.1.2 Software Analogies

To better understand how HDL interfaces could be improved, it is useful to draw comparisons with software design, which has had to deal with similar issues. We now explore situations in software that are analogous to the interface and control limitations described in Section 1.1.1.

For more than 30 years, compilers have facilitated the use of third-party code by standardizing a set of calling conventions between functions. This protocol is similar to intermodule communication in HDLs, in that it allows software modules to communicate with each other. These calling conventions include details about various register assignments for maintaining state on the stack (e.g., function arguments and return values).

If the handshake isn't unified across all functions, then code becomes incompatible (e.g., this can occur when functions are compiled using different compilers). Worse yet, if the compiler didn't automate the stack preparation and register assignments, each author would be responsible for implementing their own convention, and interoperability between functions would suffer significantly. Surprisingly, this scenario is most similar to the current state of digital design, which leaves everything up to module authors. Unfortunately, the most expedient solution is often an inflexible protocol, which works well for the immediate case but is generally timing-sensitive and leads to future incompatibility in other environments. To ameliorate this problem, module designers should agree on a latency-insensitive handshaking convention across

interfaces. This will allow systems to function correctly despite changes in message latencies.

Another useful software technique involves generic programming, such as templates in C++ that allow compile-time specialization and optimization. These techniques allow the same code and logic to be reused and specialized in different use cases. Note that template meta-programming in software closely resembles our desire to parameterize control logic in HDL interfaces.

Furthermore, software compilers often convert generic programs into efficient, optimized code for its environment by utilizing compile-time information in interfaces. Similarly, we want to produce efficient control logic from our parameterizations, and efficient communication from more generic interfaces.

Although initial HDL specifications lacked generic programming capabilities, more recent versions do have some limited support. For example, elaboration-time parameters in the Verilog 2001 standard [17] allow limited functional changes during compilation, but also contain a variety of shortcomings in their expressivity and typing [30]. More complex elaboration parameter types will be needed to facilitate better compile-time flexibility in control logic.

1.1.3 Why Are HDL Interfaces Inflexible?

We have argued that HDL interfaces should be more generic to allow better module reuse, and even noted that the software community has addressed and solved similar issues. This begs the question as to why HDL interfaces have remained inflexible. To address this, and to better understand why hardwired interfaces are still used in HDLs, we now discuss fundamental differences in design constraints between hardware and software.

Many software applications can tolerate a significant amount of overhead (as extra instructions, wasted cycles, etc.) without any perceptible difference to the user-experience. As computer hardware continues increasing in performance, software can generally become less efficient without serious side-effects. Hence, higher-level software languages are constantly being adopted, since designers are happy to trade

design time for efficiency. Newer dynamic languages are often preferred for many applications with acceptable performance, gladly sacrificing runtime efficiency for the benefit of greater code flexibility and more reuse.

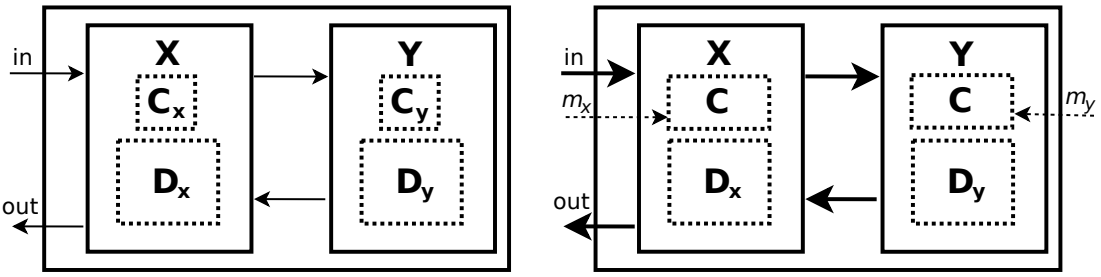
In contrast, hardware faces a renewed focus on energy-efficiency due to physical limitations of technology scaling[32]. Moreover, many modern hardware applications are battery-powered (e.g., mobile phones), and hence very sensitive to energy consumption. For these reasons, hardware designers are more adverse to overhead than software designers, and so they are less willing to adopt higher-level techniques that sacrifice efficiency. Indeed, as we will show in Section 3.1, current logic synthesis tools have limitations when compiling flexible designs, suggesting why hardware interfaces have remained inflexible.

Naturally, this leads to the question of whether it is possible to use more generic interface abstractions in our designs while reliably removing any overhead when they are synthesized. The following section summarizes the properties that high-level interfaces should possess. Later, Chapter 2 shows how to build them, and Chapter 3 addresses the removal of overhead from compiled high-level interfaces.

1.2 Proposal: High-level Interfaces

We now present high-level interfaces, built on top of RTL, as enhancements to overcome the limited flexibility discussed in Section 1.1. Figure 1.2 highlights the differences between a conventional RTL design and one using high-level interfaces. Instead of communication using fixed, rigid connections, modules should agree upon more flexible communication protocols, allowing them to pass messages in a latency-insensitive manner. This will greatly facilitate module reuse and refinements by decoupling intermodule timing dependencies that are commonplace in RTL.

Moreover, high-level interfaces abstract control-flow logic into interface elaboration parameters. By facilitating compile-time control changes, modules can operate more flexibly, covering a wider variety of conditions. Implementing different control procedures no longer requires creating and maintaining separate RTL for each desired implementation case. Instead, one generic control module can be used everywhere,



(a) An example of a typical RTL design consisting of 2 modules, X and Y.

(b) The same design using high-level interfaces. Note that control logic has been abstracted out, so the same generic control block (C) can now be used in place of specialized control blocks, and programmed with elaboration parameters m_x and m_y. Thicker intermodule connections indicate improved latency-flexibility.

Figure 1.2: Comparing a conventional RTL design to one augmented with high-level interfaces. Logical blocks are separated into data-path (D) and control-path (C).

reducing RTL code complexity.

Although high-level interfaces offer a promising value proposition for designers in terms of design-time flexibility, it is important that they don't contribute overhead compared to pure RTL (as discussed in Section 1.1.3, hardware designers tend to be sensitive to overhead). Therefore, while high-level interfaces are a nice design-time abstraction, we need to ensure that they synthesize to gate-level netlists that are similar to handwritten RTL in terms of energy, area, and performance. In other words, we want the designs in Figure 1.2a and Figure 1.2b synthesize to equivalent netlists. By ensuring high-level interfaces introduce minimal overheads, we make a much stronger value proposition and encourage their adoption.

1.3 Related Work

There have been numerous prior efforts to improve the RTL abstraction and increase designer productivity. This section explores some of this previous work, and describes how it relates to our concept of high-level interfaces.

1.3.1 Construction with Meta-Languages

Vanilla RTL lacks features for flexible compile-time module construction, so it did a poor job at producing reusable code. To improve upon this, the Verilog 2001 standard [17] introduced integer elaboration parameters and *generate* loops, allowing signal-widths and sizes of data-path arrays to be varied at compile time. While this was a great improvement for code reuse in certain cases, it does not allow more complex elaboration parameter types (e.g., 2-dimensional bit arrays) and is still limited to Verilog syntax within *generate* statements.

More recently, SystemVerilog [18] includes support for 2-dimensional bit arrays as elaboration parameters. However, the syntax for specifying array values is cumbersome. Moreover, tool support of this feature seems to be extremely limited.

To combat the limitations of this elaboration/generation, people have built tools that allow RTL to be generated using meta-languages that are more flexible than Verilog. For example, Genesis2 [30][33] uses a Perl preprocessor to generate RTL, and supports hierarchical complex parameter types, including 2-dimensional arrays and associative arrays. Similarly, Chisel constructs RTL using Scala[3]. Since widespread support for complex parameters remains limited in RTL languages, these meta-language constructors are convenient for realizing the control-logic abstraction required in high-level interfaces.

1.3.2 High-Level Synthesis

In addition to using meta-language constructors, some have proposed entirely new languages in which designs can be described. These high-level synthesis (HLS) languages are usually automatically compiled to RTL, and examples of these languages include SystemC [1] and BlueSpec [26]. These are distinguished from meta-language constructors because they move beyond flexible code generators, representing entirely new languages designed to fully capture algorithm semantics.

By fully capturing algorithm behavior, HLS compilers can theoretically generate required control logic that would otherwise need to be written explicitly. Examples of this include logic for scheduling and sharing of resources. HLS has the potential

to free designers from having to worry about low-level details (e.g., bits and wires) as compared to RTL. In theory this not only makes it easier to code individual modules, but also makes it easier to explore and compose new designs because of the increased automation and higher abstraction.

Although HLS has many potential advantages over RTL, its adoption has been slow and RTL remains the predominant design abstraction used in industry. Most HLS work has been somewhat domain specific, focusing on describing flexible datapath elements (e.g., for implementing signal processing algorithms), and the required control logic to make them work. As HLS becomes more mainstream, however, there will be a need for more general and flexible interfaces between modules, so the ideas in this work are complementary to many of the potential HLS advantages. In theory, HLS frameworks could automate the use of flexible communication protocols by generating the required interface logic around modules[1]. Note that regardless of whether high-level interfaces are embedded in HLS frameworks, or implemented manually in RTL, the optimization strategy in Section 3.2 will still be required to compile interfaces into efficient implementations.

1.3.3 SystemVerilog Interfaces

The recent introduction of SystemVerilog *Interfaces* allowed definitions and directions of interface bits to be consolidated in one place, instead of requiring this information to be redundantly stored in multiple module definitions. While this removes some of the tedium in RTL module definitions by moving toward a don't-repeat-yourself (DRY) design pattern, it does not address the more important issue of the actual communication protocol. High-level interfaces build upon these signal definitions, allowing each signal to be sent as a latency-insensitive message. As we show in Section 2.1.1, our latency-insensitive handshake implies additional interface bits alongside each module input and output signal.

[1] One unpublished demonstration of these automatic wrappers used BlueSpec[41].

Chapter 2

Building High-Level Interfaces

Hardware description languages (HDLs) and register-transfer level (RTL) logic and have played crucial roles in the design of digital systems throughout the last 20 years. They enable designers to work with higher-level logic representations instead of transistors or logic gates, which dramatically increases designer productivity. Since these higher-level representations can be automatically converted into efficient lower-level gates and wires (via synthesis, place, and route), their adoption has been ubiquitous.

Despite their widespread prevalence and continued success, conventional HDLs have a number of shortcomings in their ability to produce *flexible* designs. Cross-module communication is brittle, and correct system operation generally depends on specific module timings. This makes it difficult to both refine a particular module within a design and reuse a module in a different design. Moreover, control logic within modules is fixed in the code, making it difficult to tweak functionality for different use cases.

High-level interfaces are abstractions that provide additional timing and functional flexibility in RTL[1]. Note that high-level interfaces do not represent specific physical structures, but instead refer to two key properties that module interfaces should have: (1) latency-insensitive handshaking, and (2) parameterized specialization (particularly control logic).

[1]Although this work focuses on RTL, these same ideas can be incorporated in high-level synthesis frameworks, and will still benefit from the optimization techniques in Section 3

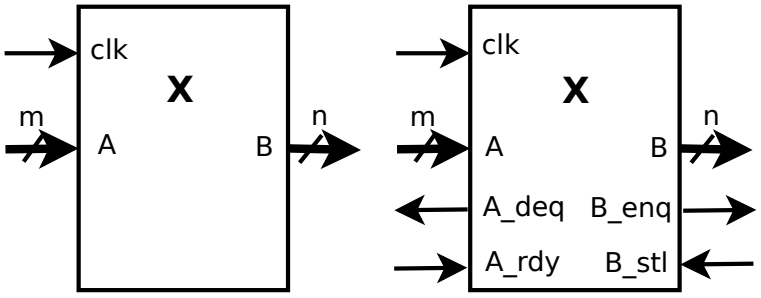
To address the former, we present a latency-insensitive communication protocol, whereby modules can pass messages in a timing-independent (logically asynchronous) fashion. To address the latter, we can utilize microprogrammed controllers in lieu of conventional hard-coded finite state machines[19]. By using these control structures, we can pass the control program as a complex elaboration parameter, allowing per-instance control flexibility. This chapter describes these design styles in more detail. Chapter 3 will discuss the sources of overhead that arise from these techniques in today's design process and show how most of it can be removed.

2.1 Latency-Insensitive Handshaking

The goal of latency-insensitive handshaking is to decouple a module's timing from its functionality. Using handshaking protocols, a system will function if the modules maintain the right order of the messages sent on the links, and not depend on strict timing. The end result is two-fold: a system that is easier to modify since changes in timing don't affect functionality, and modules that are easier to use in different environments, since they have fewer environmental assumptions.

Without strict latency-insensitive handshaking, it is all too easy for cross-module timing dependencies to creep into designs (even inadvertently). In fact, this is what most designers have been trained to do (and it's currently the most energy efficient approach). These dependencies are completely benign except that they lead to hard-to-modify, timing-inflexible systems. By employing a strict latency-insensitive handshaking protocol across interfaces, system designers can ensure these cross-module timing dependencies do not inadvertently creep into the design.

There are a variety of ways to build systems with latency-insensitive interfaces, from disciplined conventions in RTL to automatic high-level synthesis transformations. While these different approaches certainly have tradeoffs, this work focuses on their similarities rather than differences. Compared to regular designs, latency-insensitive designs generally require additional storage elements (e.g., FIFOs) as well as modified control logic (e.g., stalls) to account for different latency behaviors. Regardless of how the interfaces are constructed, these additional elements create timing,



(a) Original interface (b) Latency-insensitive interface

Figure 2.1: Converting the interface of module X to implement a latency-insensitive handshake

area, and energy overhead as compared to a hand-optimized design. Section 4.4 will show examples where this overhead from a single interface can affect an entire design's area by 20%.

Note that similar ideas of flexible communication are widespread at the macro level of chip design. For example, complex SOCs often have buses or on-chip interconnection networks to facilitate connecting various IP blocks. Although these structures usually require extra energy and/or cycles compared to a hard-wired approach, this overhead is generally acceptable given the obvious design benefits.

In contrast, latency-insensitive handshaking protocols should also be used across lower-level microarchitectural blocks, where their overhead can be significant. Previous work has proved that latency-insensitive IP blocks can be correctly composed into complex digital systems [6] [5] [7] [42], but these works generally ignore implementation overhead. In Section 3.1 we will discuss this implementation overhead, and in Section 3.2 we will show optimization techniques that can remove it.

There are many possible protocols to guarantee latency insensitivity. The following subsections describe the one that we will use throughout this thesis.

2.1.1 Handshake Protocol

The protocol we will use adds two ports for each output port (a pulse-based output *enq* and a level-based input *stl*), and two ports for each input port (a pulse-based

output *deq*, and a level-based input *rdy*). Figure 2.1 depicts these additional interface signals. This protocol requires a FIFO between modules to store messages when they are not ready to be consumed, giving timing flexibility across interfaces.

enq ("enqueue")

Each logical output should be accompanied by an additional 1-bit pulse-driven output signal *enq*. Each *enq* pulse indicates that the associated output data is in a valid state for that clock cycle, and is used as a write-enable by the FIFO. It is illegal for a module to assert *enq* when the associated *stl* (backpressure) is active. This mechanism ensures output data will be consumed downstream without requiring any acknowledgement.

stl ("stall")

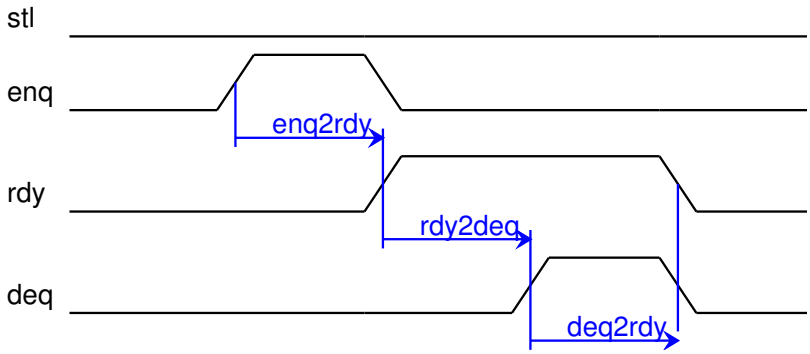
Each module output should have an associated level-driven backpressure input *stl*, which is driven by the FIFO. In the simplest case, *stl* can be a 1-bit "stall" signal to indicate the FIFO is full. In general, it can be a multi-bit "credit" signal, indicating the number of remaining *enq* pulses that can be safely produced until the FIFO is full (requiring a stall). For the system to function correctly, a module must never assert *enq* when its associated *stl* is active. This allows the system to function with modules that consume more slowly, avoiding potential overflows of buffer space.

deq ("dequeue")

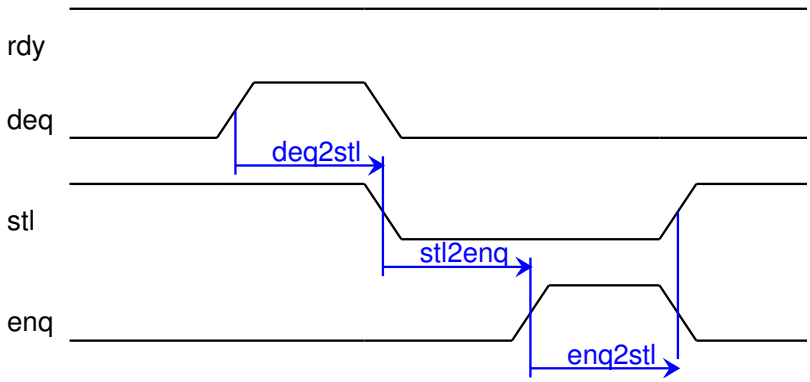
Each logical input should include a 1-bit pulse-driven output signal *deq*. Each *deq* pulse indicates to the FIFO that one token of input data is being consumed. It is illegal for the *deq* signal to be high when *rdy* level is low.

rdy ("ready")

Each logical input should include a 1-bit level-driven input *rdy*, driven by the FIFO. When *rdy* is high, there is new data on the interface ready and waiting to be consumed.



(a) Timing diagram for a relatively slow producer.



(b) Timing diagram for a relatively slow consumer.

Figure 2.2: Timing and transitions of a latency-insensitive handshake.

Timing

Figure 2.2a depicts the timing dependencies of these signals for a slow producer, and Figure 2.2b depicts them for a slow consumer. If the depicted latencies represent logical clock cycles (i.e., they are non-negative integers), then we require $rdy2deq + deq2rdy \geq 1$ and $stl2enq + enq2stl \geq 1$ to prevent combinational feedback loops. All other latencies may be 0 or more cycles.

2.1.2 Bypass FIFO

Assuming all modules have implemented the protocol modifications described above, it is straightforward to connect them with bypass-enabled FIFOs. These FIFOs act as

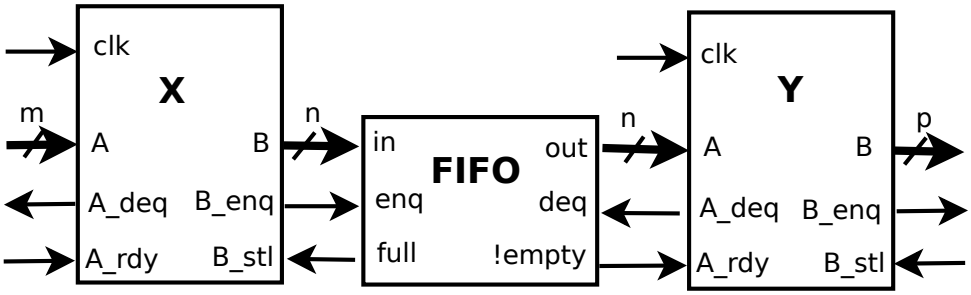


Figure 2.3: Connecting two modules X and Y with a bypass FIFO for latency-insensitive communication.

distributed buffer space for messages between modules, and allow for desired latency isolation. See Section A.1 for a reference Verilog implementation of such a bypass FIFO.

The *enq* and *deq* pulses directly connect to the FIFO's interface. The *stl* level comes directly from the FIFO's credit counter (or *full*) signal. Likewise, the *rdy* level comes from the FIFO's *empty* output. Figure 2.3 depicts this connection.

A bypass FIFO has a combinational path from input to output, and sets the latency $enq2rdy = 0$. When converting a hard-wired interface to a latency-sensitive handshake, this property ensures the handshake does not add extra cycles of overhead. [2] This property proves useful when converting hard-wired links to be latency-insensitive, since it is generally easier to debug a system when a cycle-accurate golden model is available. Furthermore, the presence of a combinational bypass path facilitates reachability analysis: it allows the technique discussed in Section 3.2 to prove when the bypass path will always be active, allowing FIFO overhead to be safely removed.

2.1.3 Important Issues

The previous sections explored how to construct a functioning latency-insensitive link between two modules. However, there are a number of important considerations when modules with these interfaces are used to compose larger systems. Note that

[2]Note the combinational bypass logic does add cycle-time overhead, but that is ignored here.

some issues are only relevant when modifying existing functioning hard-wired systems, while others are more pertinent when composing systems from scratch. There is not one universally superior approach to dealing with each of these issues. The various ideas discussed in this subsection will give insight about possible approaches, but we ultimately leave it to the designer to weigh the pros and cons of each approach for their environment.

Combinational paths

Although latency-insensitive interfaces facilitate creating new systems by composing existing modules in new ways, care must be taken not to inadvertently introduce excessive or illegal combinational paths, particularly when composing new systems. It is important to note that the interfaces do not force timing isolation between modules because of the combinational bypass paths. Hence, placing many combinational modules in series will still create long combinational paths that may make timing closure difficult. Moreover, connecting combinational modules in a feedback loop will create combinational loops, which are illegal in standard-cell designs.

Designers can protect against these issues by adopting strict conventions to ensure they never arise, or simply by solving them on a case-by-case basis. Note the long-path issue is not new, and the same solution of adding internal pipeline registers applies. Similarly, the feedback issue can only be solved by adding a register somewhere in the loop. These observations suggest that all modules implementing latency-insensitive interfaces may want to include optional internal pipeline registers, so that they can be used if needed. This is similar to common system design conventions that require modules to uniformly agree on registering all inputs (or all outputs).

Sizing FIFOs

It is important that the FIFOs are sized to have appropriate depth. If they are too small, then they will quickly fill with messages, asserting backpressure stalls and causing system performance to suffer. Moreover, under-sized FIFOs can cause deadlocks in some bounded dataflow networks[42]. Conversely, if the FIFOs are too large, then

the implementation will contain unneeded FIFO space, contributing area and energy overhead. In many practical situations, single-element FIFOs are sufficient.

As we will see, the reachability method in Section 3.2 can determine which registers in a design are used and which are not, allowing unneeded ones to be removed. This suggests an approach where a designer initially oversizes FIFOs, and then relies on our optimization technique to prune them if possible.

Deadlock

Deadlock in a network of latency-insensitive modules is a situation where one or more modules are stalled indefinitely, either waiting for a new message to arrive (*rdy*) or waiting for a downstream buffer to clear (*stl*). Note that we ignore other forms of deadlock that can occur in systems without latency-insensitivity (e.g., protocol deadlock), as they are beyond the scope of this work. Since deadlock often causes a catastrophic system failure, it is important for designers to be aware of how it can happen, and how it may be avoided. In practice, a number of approaches have been used for avoiding deadlock, ranging from ad-hoc detection and prevention in simulation to rigid design conventions that guarantee a network is deadlock free.

The necessary condition for deadlock in a system with latency-insensitive interfaces is a cycle, or loop, in intermodule communication. If there are no cycles, deadlock while waiting on the interfaces need not be considered. This situation commonly occurs in pure dataflow pipelines (which have no feedback) that are constructed with latency-insensitive modules, as well as in hard-wired systems where only 1 critical interface is converted to be latency-insensitive (since a cycle requires at least 2 interfaces).

For other designs that do have circular interface dependencies, we know that deadlock *can* occur, but it is still not guaranteed. Prior work in bounded dataflow networks showed that deadlock cannot occur in a network of latency-insensitive modules, as long as each module has no extraneous dependencies (NED) and is self-cleaning (SC)[42]. The NED property dictates that each module output should only wait on inputs that it *needs* (whereas a naive approach might make each output wait on *all* module inputs). The SC property dictates that there must be a 1-to-1 correspondence

between enqueued outputs and dequeued inputs. That is, given that an output has been enqueued, the corresponding inputs must be dequeued at some point (either before or after the output is produced). Typically, modules will dequeue inputs, do their computation, and later produce outputs, satisfying the SC property.

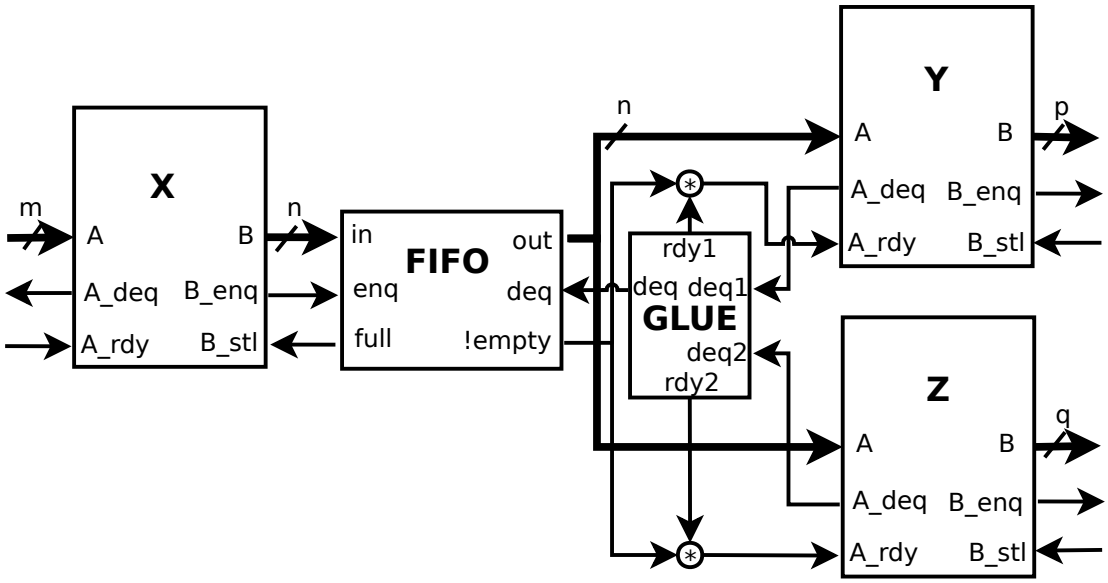
Deadlock is a serious concern for all complex systems, regardless of whether they use latency-insensitive interfaces. However, if a system with hardwired interfaces is deadlock-free, then in our experience adding latency-insensitivity is unlikely to introduce new deadlock situations. While enforcing all modules to obey the NED and SC properties may guarantee deadlock-free operation, these constraints may not always be natural or practical in real settings (particularly when converting existing RTL modules). One such example is shown in Figure 2.4b, where the simpler design choice violates NED. We believe the best approach for deadlock prevention is a mixture of awareness, common-sense, and (as always) rigorous testing and validation.

Intermodule Forks

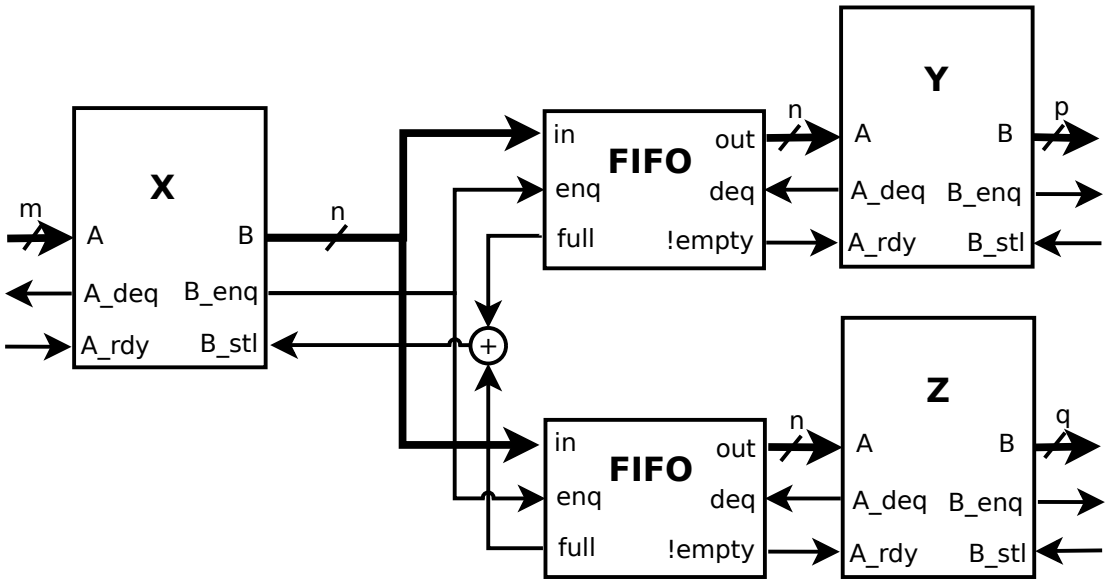
In most systems it is common for a module output to be used as input in more than one module. In these intermodule forks, the simple interface depicted in Figure 2.3 is not applicable. Instead, there are two distinct approaches we can take, depicted in Figure 2.4.

Figure 2.4a shows an example of sharing the FIFO among all modules. For this to work properly, additional logic in *GLUE* is needed to combine the individual *deq* pulses so that only one pulse ever reaches the FIFO. When only 1 (but not both) of Y or Z has fired *deq*, it must be stalled (by setting *rdy1* or *rdy2* to 0) until the other has fired *deq* as well. Note the *GLUE* module can be implemented by a 4-state FSM. In contrast, Figure 2.4b shows how the FIFOs can replicated for each module input. In this example, the *full* signals must be combined with a logical OR before producing *stl*. Since it is generally simpler to combine levels than pulses, the *full* signals can be combined using a simple OR and does not require an FSM.

These two possibilities offer different tradeoffs for designers. The shared FIFO approach uses less FIFO space, but requires considerably more glue logic. The separate FIFO approach provides better timing isolation between Y and Z, which can offer



(a) Forking with a shared FIFO. The *GLUE* module represents FSM logic needed to combine the two *deq* pulses into one pulse, as well as to appropriately stall modules while waiting for others.



(b) Forking using separate FIFOs. Note the FIFO *full* levels are OR'd to create the *stl* level, creating an extraneous dependency among Y and Z.

Figure 2.4: Possible latency-insensitive interface implementations when 1 output forks into 2 or more inputs.

better system performance and flexibility at the cost of additional FIFOs. Note the separate FIFO approach creates false dependencies between the outputs, which violates the NED property previously discussed, but it can still be advantageous in many cases. Since our optimization method in Section 3.2 can remove unneeded FIFOs, we favor the separate FIFO approach.

Intramodule forks

There can also be intramodule forks, where one input is used to generate two or more outputs. Note that although this situation has similar constraints (and similar solutions) to the intermodule forks previously discussed, we mention it separately because it often arises differently. While intermodule forks typically appear appear in system design when connecting latency-insensitive blocks, intramodule forks are more likely to arise during module development, and may even influence module organization.

These "shared input" cases require additional internal logic to ensure the latency-insensitive protocol is obeyed. One simple approach is to stall until the logical OR of all *stl* inputs is low. Although this violates the NED property previously discussed (since outputs are then dependent on other outputs), it can still be useful in many cases, and ensures the input is only dequeued once. Another approach is to introduce complex pulse logic to ensure the input is only dequeued once, similar to the function of the *GLUE* module in Figure 2.4a. In other cases, it can be better to reorganize the larger module into separate modules of one output each, transforming the problem into an intermodule fork.

Latency-insensitive handshaking protocols are useful because they yield intermodule flexibility. This facilitates design-space exploration and application-specific refinements by allowing different module instances to have varied timings, but does not address how these instances should be built. A naive approach is to build independent instances, but this is often overkill as these variations can sometimes be realized

with simple modifications to control-logic. To address this, the following section presents an abstraction to design flexible controllers, complementing the intermodule flexibility of latency-insensitive protocols by providing *intramodule* flexibility.

2.2 Flexible Controller Design

Historically, microprocessor designers moved toward reconfigurable controllers to simplify their lives. Instead of needing to create hardwired logic, they were able to focus on writing microcode, a series of simple microinstructions that are loaded into a specialized memory at boot-up, and fetched, decoded, and executed during normal operation[43][40]. This improved abstraction simplified design and facilitated changes late in the design process. Later implementations even had writeable control stores, allowing in-situ modifications and bug-patching[28][15]. Although these microprogrammed control-store implementations require more area and energy to operate than their hardwired counterparts, microprocessor designers were (and continue to be) willing to make this tradeoff.

In practice, many modern ASICs use a combination of microprogrammed and hardwired control. The microprograms are generally used for higher-level tasks that can accommodate more overhead (and that may require in-situ patching), while hardwired logic is used for lower-level control that demands more efficiency. Although it is used for relatively simpler controllers, such hardwired logic in RTL is still difficult to tweak and modify during the design process, making module reuse more difficult.

High-level interfaces allow this hardwired control to instead be expressed as microprograms that are still compiled into hardwired logic, providing the design-time benefits of microprogramming without the runtime overhead. By utilizing the same microprogram abstractions as microprocessor designers, high-level interfaces can leverage the same prior work and tools that have been developed to write microcode. The difference is that instead of loading a microprogram into a dedicated memory at boot-up, the microprograms will be specified as elaboration parameters at the interface, and compiled into efficient hard-wired logic during synthesis. This section describes reconfigurable controller design in more detail to better understand these

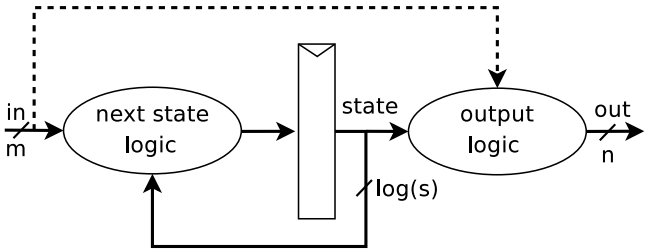


Figure 2.5: A generic finite state machine with m inputs, n outputs, and s states. Output logic may or may not depend on the input according to style. Note required storage element.

control abstractions.

2.2.1 Combinational Logic

We begin by quickly reviewing configurable combinational logic because (as detailed in Section 2.2.2) it is the fundamental building block of reconfigurable controllers. An arbitrary boolean function can be implemented by storing the function's truth table in a programmable memory, and addressing the memory using the function's inputs. In this setup, an arbitrary function with m inputs and n outputs can be implemented in a memory of width n and depth 2^m. We note that such structures are common and can be found in designs under a variety of different names, such as programmable decoders, ROMs, and lookup tables (LUTs) in FPGAs[44].

2.2.2 Finite State Machines

Finite state machines (FSMs) are a convenient abstraction that helps in the design of simple controllers. These sequential control circuits are characterized by a finite number of internal states, state transitions, and outputs. They are typically represented as finite state diagrams, which depict the various states and state transitions. Fig. 2.5 shows a generic s-state FSM hardware implementation, in which state transitions depend on the current state as well as current inputs, and outputs depend on the current state and (depending on style) inputs.

The ability to design flexible FSMs is particularly relevant for chip generators

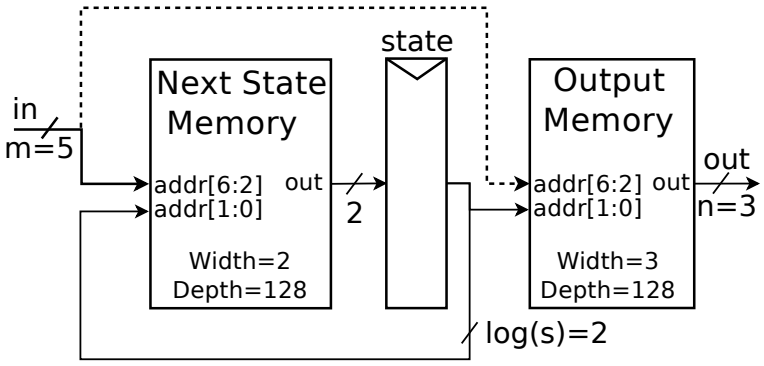


Figure 2.6: A 5-input, 4-state, and 3-output FSM implemented with asynchronously readable memories.

because FSMs are the brains behind hardware operation, so flexible FSMs enable different operational modes within one larger framework. A reconfigurable FSM can be realized by using programmable tables to implement its combinational logic bubbles (both next-state and output). For example, Fig. 2.6 shows how a 4-state FSM with 5 inputs and 3 outputs can be implemented with two memory elements: a 2-bit-wide next-state memory with 2+5=7 address bits (128 entries), and a 3-bit wide output memory also with 2+5=7 address bits (128 entries).

Microcode sequencers are FSMs whose conceptual operation is described by microprograms instead of finite state diagrams. Microprograms are a series of simple *microinstructions,* low-level operations that assert particular control signals on a given cycle. We refer to the bit-level representation of microinstructions as microcode. Due to their sequential nature (as well as their resemblance to assembly programming), many designers find microprograms to be more convenient than finite state diagrams for describing controllers, particularly as the design complexity grows. In practice, microcode format varies from being inefficiently encoded (known as horizontal microcode) or efficiently encoded (vertical microcode), allowing a tradeoff in decoder complexity. Many microprogramming systems employ horizontal formats to simplify the paths between the controllers and the datapath units [29], using separate subfields to control different units in the design.

Despite their different controller abstractions, the operation of programmable FSMs and programmable microcode sequencers turns out to be similar. Figure 2.7

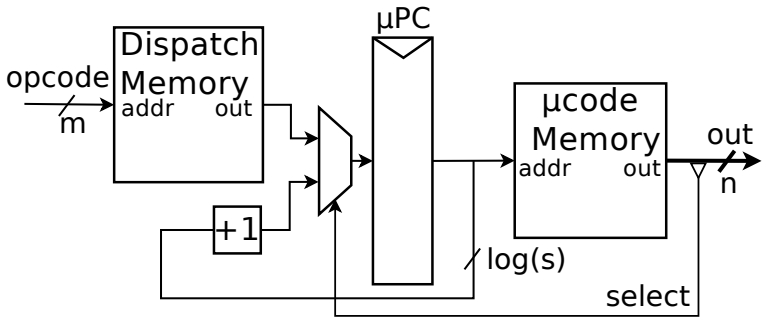


Figure 2.7: A microcode sequencer supporting generic non-conditional dispatch functions. Note the structure is similar to Figure 2.6, except the *Next-State Memory* has been replaced with an incrementer and a *Dispatch Memory* to handle branches. This alternative structure often leads to smaller implementations.

shows the hardware implementation of a typical microcode sequencer, which resembles the FSM implementation in Fig. 2.6. Note the microcode memory performs similarly to the output logic of FSMs, and the primary difference is the next-state logic. In FSMs, the next-state logic is fully general, allowing direct transition from any state to any other state. In microcode sequencers, on the other hand, the expected transition is a trivial increment to the next sequential microprogram counter. Other state transitions (jumps) are flagged and handled by dedicated dispatch tables, which tend to be small for many practical designs. For these reasons, microcode sequencers are often a more efficient way to implement runtime reconfigurable controllers. For purposes of pre-silicon (design-time) reconfigurability, however, we do not need to make significant distinctions between FSMs and microcode sequencers, because they both share the same underlying table-driven logical descriptions. For these reasons we will use the terms "microcode sequencer" and "table-driven controller" synonymously.

The table-driven representation for controllers has a number of advantages. It facilitates patches late in the design cycle, and writeable control stores were shown early-on be an effective tool for tuning processor performance in certain applications[4]. Sorin et al. argue that a single table-driven approach can be used in many design phases, including specifying, documenting, and verifying cache coherence protocols[34]. Firoozshahian et al. go a step further and describe how programmable, table-driven controllers can allow a memory controller to support different memory models and

protocols within a CMP system[14]. However, these table-driven implementations come with significant area and cycle-time costs from the added memories and address decoding logic. Our desire to leverage many of the advantages of microcode-based controllers, while achieving implementation efficiency, naturally leads to the question of whether we can produce efficient controller implementations from these microprograms. The optimization methodology that will help us achieve our goal is broadly known as *partial evaluation*, and is discussed in Section 3.1.

Chapter 3

Removing Overhead From High-Level Interfaces

Despite their advantages, high-level interfaces are not often used in HDL designs. As we will see, the main issue is the implementation overhead associated with flexible components. Intuitively, there is always a tradeoff between flexibility and efficiency: a module that operates correctly across multiple conditions inherently has more states and more logic than its customized counterpart. A system designer, knowing the system timings, wants to build and use components tailored for that application, and thus avoid paying for extra area, energy and performance overhead. But what if the overhead could just "go away" all on its own? After all, in theory a logical function doesn't depend on the way it is coded. We begin this chapter by studying the efficacy of modern VLSI tools at removing the overhead automatically. This study reveals that, unfortunately, modern synthesis fails to propagate reachability information across flop boundaries, thus preventing efficient removal. We then suggest supplementary techniques to overcome this limitation, allowing most of the remaining overhead to be identified and removed within the context of a standard tool flow.

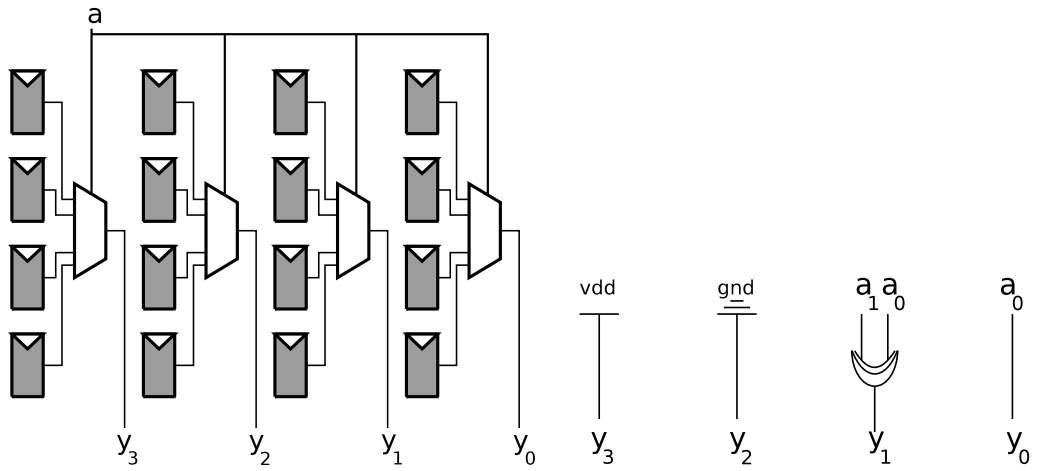
3.1 Partial Evaluation with Current Tools

To evaluate current synthesis tools, we measure how well they synthesize various compile-time flexible structures (compared to equivalent inflexible/custom implementations). We do this by "programming" reconfigurable tables with constant values, and allow the tool to infer additional optimizations based on these constants. Since combinational logic optimization is a well-studied topic, we expect synthesis tools to do this well (and, as we will see, they usually do).

This technique, broadly known as *partial evaluation,* has been used to specialize generic software programs for years. It uses known information about program inputs at compile-time to reveal new optimizations that were previously unavailable, allowing the compiler to produce better code. This methodology lets programmers write broad general-purpose programs that then compile into specific optimized code instances. The C++ Standard Template Library (STL) is a common software implementation that relies on partial evaluation [36].

Despite its prevalence in software, partial evaluation (PE) methodologies in hardware design have been primarily limited to data-path optimization in domain-specific frameworks. McKay et. al. apply PE to FPGA synthesis of generic data-path elements for DSP chips [22]. Leonard and Mangione-Smith apply PE to a DES algorithm where the secret key is known and fixed [21]. Mukherjee and Vemuri use PE to optimize DSP data-path elements at the transistor level [25]. Our work extends this strategy to include control-path elements as well as data-path elements. Not only do we want efficient functional (data-path) units, but we want to efficiently control them in different ways, and by doing so we enhance our ability to build useful flexible modules.

Figure 3.1 depicts partial evaluation of a 4x4 array, an optimization that will be performed by most modern tools. Note that for simplicity only the asynchronous read logic is shown and the write logic is ignored. If the array values are fixed, then not only are the state flops and the addressing write logic removed, but the 4-to-1 read multiplexers are simplified. Assuming the array is fixed with the values in Table 3.1, the read logic is simplified substantially to that shown in Figure 3.1(b). In other



(a) A 4x4 LUT implemented as an addressable flip-flop array. The address bits, a_i, determine what line to assert on the output, y.

(b) Optimized with the values in Table 3.1

Figure 3.1: Partial evaluation example of a 4x4 lookup table (LUT)

Table 3.1: An example set of values for a 4x4 array

Address	b_3	b_2	b_1	b_0
00	1	0	0	0
01	1	0	1	1
10	1	0	1	0
11	1	0	0	1

words, the lookup table is converted back to the logical function it implements.

In general, for partial evaluation of reconfigurable controllers to be effective, we desire the optimized controller to approach the area and timing efficiency of a directly implemented (non-programmable) controller (similar to the example in Figure 3.1). Our hand-tuned results in the following subsections explore this tradeoff between full-custom and automatically-optimized circuits. In our experience, a synthesis compiler needs a few key optimization techniques before it can properly perform partial evaluation of table-based structures. Beyond standard logic reduction methods, these techniques include the ability to identify any known restrictions that might simplify a signal state (thus, a non-optimally encoded signal), propagate these restrictions downstream, and perform typical logic optimizations using this state information.

We note that it is not uncommon in large designs to find signals that are not encoded optimally, either intentionally, for instance to reduce the need for decoding logic by storing fully decoded fields in horizontal microcode, or unintentionally, such as occurs when reusing generic modules. We will refer to the downstream propagation of signal restrictions as *state propagation*, and the logic optimizations that use this information as *state folding*. Note that these terms are analogous to the familiar software compiler terms *constant propagation* and *constant folding*.

More formally, an n-bit signal y has $k = 2^n$ possible states in a physical design: $y \in \{0, 1, 2, 3, ..., 2^n - 1\}$. If we know of any restrictions on y, then $k < 2^n$. For example, if we know that y is one-hot encoded, then we know $y \in \{1, 2, 4, 8, ..., 2^{n-1}\}$ and $k = n$. If y is used in a downstream ones-counter circuit, the compiler can evaluate all n values of the circuit and infer that the output is a constant 1, allowing the ones-counter logic to be removed altogether. This technique reduces to *constant propagation* and *constant folding* when $k = 1$.

We now turn to the practicality of design by partial evaluation; that is, we explore the efficacy of modern synthesis tools to produce optimized controller implementations from generic microcode specifications. We first compare optimized table-based implementations with fixed non-programmable implementations to confirm expected logic optimizations and the practicality of using microprogram specifications (or, more generally, tables) with high-level interfaces. We then highlight some limitations of this approach that affect both non-optimally encoded wide microinstruction formats and specialized controllers with unreachable states. We conclude by evaluating these techniques on the Smart Memories protocol controller *PCtrl,* a large microcoded design.

We chose Synopsys Design Compiler D-2010.03 to synthesize our designs as it is an industry standard tool, but we have observed similar results with other tools. The designs were coded in SystemVerilog and the synthesis library used a 90nm TSMC process.

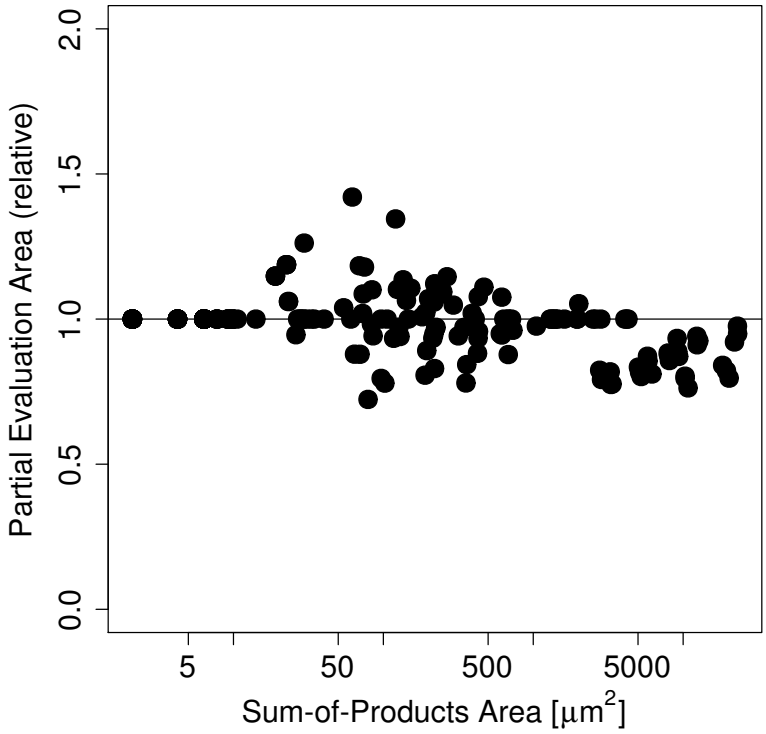


Figure 3.2: An area comparison of combinational logic synthesis results for various random designs. Note the horizontal equal-area line.

3.1.1 Constant Propagation and Folding

We start with the reconfigurable control structures described in Section 2.2 and demonstrate how closely they synthesize to their ideal directly-implemented counterparts when relying on simple constant propagation and folding. We wrote reconfigurable versions of each component using SystemVerilog. Python scripts then generated random configuration parameters for these reconfigurable designs, as well as the corresponding direct Verilog implementation for each. We then synthesized these pairs of designs over a sweep of achievable timing targets to generate synthesis results for a wide variety of design sizes and topologies. Note that we only compare areas of cycle-equivalent designs that are synthesized to identical clock periods.

Table-Based Combinational Logic

Fig. 3.2 compares the area synthesis results for many different combinational logic functions (tables of depth $d \in \{2, 8, 16, 32, 64, 256, 1024\}$ and width $w \in \{2, 4, 16, 32, 64\}$). Note that d refers to the number of entries in a fully decoded table. The "direct" (hand-optimized) implementations were written using sum-of-product assignments for each output bit. In the ideal case all points would lie on the horizontal $y = 1$ line because there would be no difference between the partial evaluation of tables and the direct implementations. However, the discrete nature of the standard cell library coupled with the "bumpy" nature of the tool's optimization surface leads to various local minima, causing the tool to find similar (but not identical) designs when starting from widely different (albeit logically equivalent) RTL descriptions. In fact, we sometimes observe slightly better results for table-based representations, especially for larger functions, suggesting sum-of-product representations are not always ideal for the tool. These observations confirm our expectation that the synthesis tool is effective at partial evaluation of combinational logic tables via constant propagation and folding.

Table-Based Controllers

Fig. 3.3 compares the synthesis results for many different FSMs (inputs $m \in \{2, 8\}$, outputs $n \in \{2, 8, 16\}$, and states $s \in \{2, 3, 8, 16, 17\}$). The direct implementation was written using a series of case statements, the style recommended by the tool vendor for automatic detection and optimization of the FSM states. The flexible implementation used combinational tables as in Section 2.2.2 to describe next-state and output logic. This change in coding style prohibited the synthesis tool from automatically detecting the FSM state encodings, leading to some variance in the synthesized areas as compared to the preferred implementations (especially for $s \in \{3, 17\}$ cases, which are not efficiently coded in binary). In a second experiment we used DesignCompiler options *set_fsm_state_vector* and *set_fsm_encoding* to manually annotate the state signal of the controller for the generic designs [38]. The plot demonstrates that providing the tool with this extra information resulted in nearly

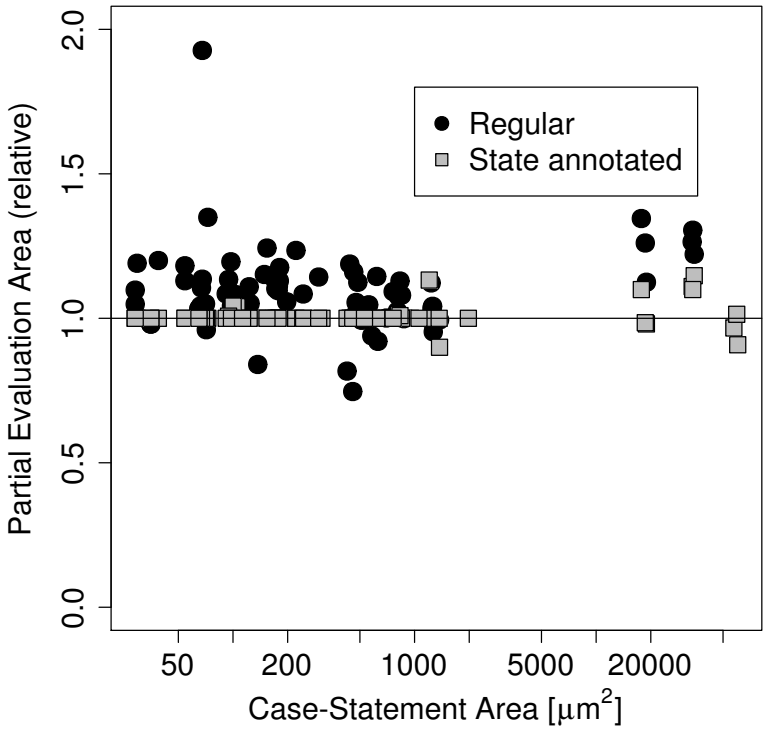


Figure 3.3: An area comparison of FSM synthesis results for various random controller designs. Note the horizontal equal-area line.

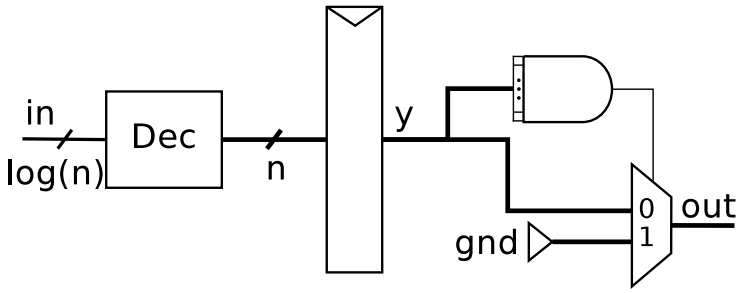


Figure 3.4: An example design to investigate state propagation and folding optimizations. Note the mux before the output is unnecessary if the signal y is one-hot encoded.

identical synthesis results between the annotated and direct implementations. It is fairly straightforward to automatically determine these state annotations from the FSM tables (or, equivalently, microcode), and so we do not see this as a real issue for abstracting state machines as elaboration parameters in high-level interfaces. Hence, we can use a flexible table-driven controller style but still achieve the synthesis benefits of a direct implementation.

3.1.2 State Propagation and Folding

Although we have demonstrated that we can achieve good implementation efficiencies for isolated controllers, we must also consider logic optimizations downstream of the controller outputs where the outputs are not fully encoded signals (e.g. horizontal microcode). This section explores the optimization of designs with k states, $1 < k < 2^n$, by examining the synthesis results of the small example design in Fig. 3.4. The one-hot decoder *Dec* allows us to specifically focus on cases where $k = n$, but we expect these results to generalize to other values of k. Note that when the signal y is one-hot, the mux on the output becomes redundant because the bitwise-AND gate should always evaluate to 0. This is the key optimization that we expect the synthesis compiler to make for this example. Although this is a relatively simple design, its synthesis properties demonstrate a number of interesting features that are consistent with our experiences on more complex designs.

We synthesized this design for a variety of different bus widths $n \in \{2, 4, 8, 16, 32, 64, 128\}$

with easily achievable timing constraints. Fig. 3.5 plots the comparative synthesis results of the generic and direct versions. The purely combinational examples (no flops) always synthesized to the ideal case, suggesting the tool correctly infers state propagation and folding in purely combinational logic. However, in the presence of flops, all of the synthesized designs failed to achieve ideal areas.

These observations suggest the synthesis compiler does not perform state propagation across flop boundaries, and cannot be trusted to consistently perform state-related optimizations. Note that we already encountered a similar situation with the states of table-driven controllers because the tool is unable to automatically recognize FSM states from tables alone. Using a hook in the synthesis tool similar to that used in the previous section, we manually annotated the states of signal y after the flop boundary, and plotted these results with filled markers in Fig. 3.5. It is clear that this state annotation allows synthesis to perform the necessary optimizations in cases where $n \leq 32$ [1]. Although horizontal microcode can be hundreds of bits, the independent subfields that drive different units tend to each be smaller than 32 bits, and so manual annotation of each subfield can still be effective. In principle, these annotations can be determined directly from controller microcrode (assuming the design has been structured properly), but in practice this can be cumbersome. Section 3.2 presents a more general approach for determining these annotations across a wide variety of designs and structures.

3.1.3 Optimizing Smart Memories PCtrl

We now examine these synthesis techniques on PCtrl, which is described more fully in Section 4.3 as an example of a large table-driven controller design. Storing all the microcode for this controller takes area, as do the associated multiplexers/decoders. To understand this overhead, we compare its original flexible design ("Full") to a partially evaluated design ("Auto") for two different memory configurations: "Cached" (a controller for local memory used as cache) and "Uncached" (a controller for local memory used as private/scratchpad memory space with no backing store). We

[1] This specific boundary at $n = 32$ is likely just a limitation of our tool.

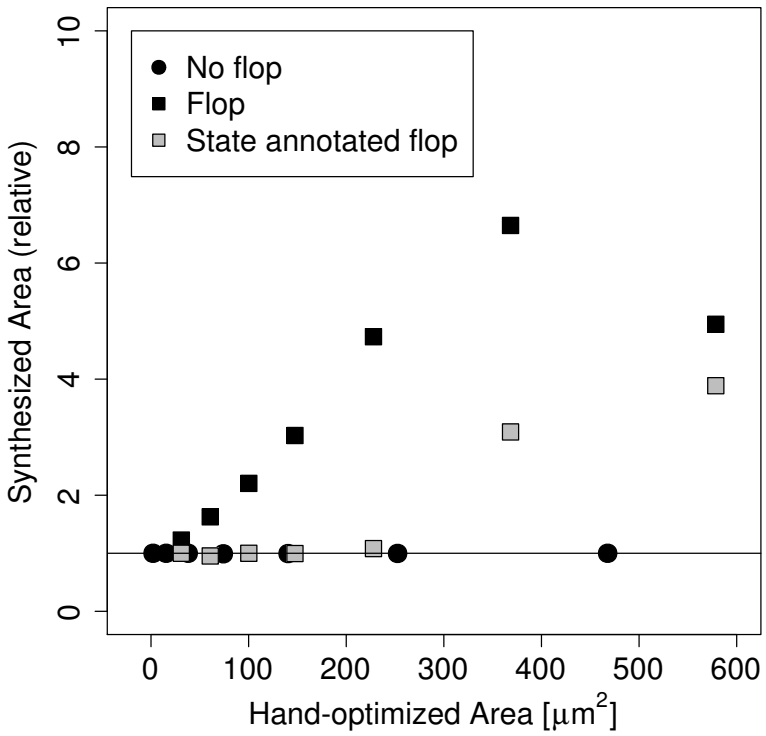


Figure 3.5: A comparison of synthesis results for the design shown in Figure 3.4. The horizontal equality line is shown.

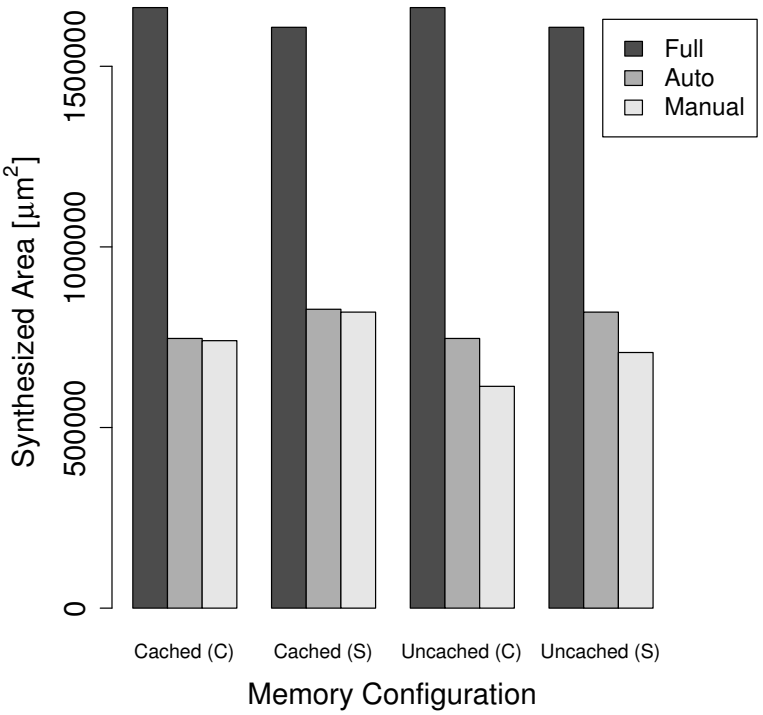


Figure 3.6: Combinational (C) and sequential (S) area usage for PCtrl instances.

further compare these with hand-optimized controller instances ("Manual") to understand the optimizations missed by automatic synthesis. Fig. 3.6 summarizes the area consumption of each design (separated into combinational and sequential logic). All designs were synthesized using TSMC 90nm technology with a 5ns clock.

The automatically optimized (via partial evaluation) controller instances halved the non-combinational area of the full design by removing all configuration memories, and halved the combinational area by simplifying access logic and propagating constants. These reductions in the controller alone represented a 7% reduction in overall chip area, which also included 8 Tensilica processors. These large area reductions reflect the conventional wisdom that runtime-reconfigurability requires significant overhead.

The differences between "Auto" and "Manual" reflect overhead missed by automatic synthesis. The manually-tuned versions include optimizations that would occur if the tool properly supported state-propagation across flop boundaries. Primarily,

these optimizations involve identifying and removing unnecessary (i.e., unreachable) addresses for specific memory modes. Since cache memory requires almost all of the original controller states, the gains from manual optimization in cached modes were minimal. In contrast, supporting uncached memory requires far fewer control states, leading manual optimization to find an additional 16% in area and power savings in the controller.

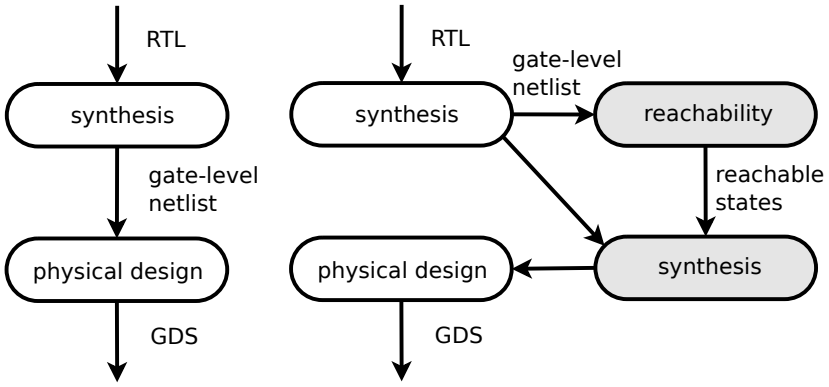
While synthesis tools worked well for this design, the results do indicate a potential issue of one moves to more generic interfaces. The Smart Memories Protocol Controller study exposed some weaknesses in the tools, which left overhead from microcode in some instances that was not accessed nor needed by the rest of the system. The following subsection explores how high-level interfaces contribute similar types of overhead. Then, in Section 3.2 we present an automated technique to overcome these issues.

3.1.4 Understanding sources of overhead

By their very nature, high-level interfaces can add many of these *unreachable* states to designs. As the simple example of Figure 3.4 showed, synthesis tools do not propagate state reachability information across sequential boundaries, thereby losing any potential downstream optimizations. As controllers are made more flexible to work correctly under more conditions, state-spaces grow to encompass a superset of all possible conditions. When instantiated in a particular environment, some of these states may never be reached, resulting in overhead. The rest of this section describes how the properties of high-level interfaces can contribute to specific unreachable state overhead. Section 3.2 discusses how this overhead can be automatically discovered and removed from designs.

Latency-Insensitive Communication

As discussed in Section 2.1, the implementation of a latency-insensitive communication protocol involves additional control states on both sides of the interfaces to handle the various possible latencies for each signal (e.g., 0 cycles, 1 cycle, or more



(a) A conventional RTL design flow.

(b) A flow modified to remove overhead from high-level interfaces

Figure 3.7: Proposed design flow modifications.

than 1 cycle), as well as the added bypass-FIFO storage element. In simple cases, a 0-cycle combinational bypass path will always be used, causing the FIFO itself and some control states to become unreachable overhead. Put another way, we don't want to pay for something that we're not using. In less trivial cases the FIFO is required, but is oversized and contains extra entries that will never be needed.

Flexible Controllers

Similarly, flexible controllers also cause overhead in modern synthesis, as shown in the experiments of Section 3.1. Overhead from unreachable controller states often manifests in downstream logic involving controller outputs. In the PCtrl protocol controller example in Section 3.1.3, we observed this phenomenon as some lookups were queued and used to address memories in other controllers in the design.

3.2 Reachability Analysis

As we showed, current synthesis methods ignore the reachability of sequential elements (flip flops). Since high-level interfaces have extra states (by design), we expect these states to be a primary source of overhead (as confirmed by the results in Chapter 4). This section demonstrates an algorithm that can identify unreachable states

in a gate-level netlist, scaling to very large designs[20]. It then shows how to annotate this information into current synthesis to improve its results. Since it operates on gate-level netlists, it can be used with either conventional HDL designs flows or high-level synthesis. Figure 3.7 depicts how our reachability technique fits into a conventional tool flow. This technique enables designers to use high-level interfaces at design-time while avoiding the current synthesis inefficiencies.

Reachability analysis is the process of identifying all legal states in a design. We use the term *sequential reachability* to emphasize our focus on sequential elements (flip flops) in designs. Reachability-related methods have been developed at many levels of abstraction, typically for formal verification of digital circuits. For example, the Murphi system uses explicit state reachability to facilitate protocol verification [12]. Other work focuses more directly on reachability in gate-level netlists for formal verification of sequential circuits [11][39][37]. Since these gate-level techniques are only concerned with proving equivalence between two designs, they can rely on symbolic equivalence checking (implicit methods) to avoid explicitly enumerating all states. In contrast, our goal is to explicitly determine the reachable design states (more similar to the Murphi approach). Implicit methods allow verification techniques to scale to larger designs because they do not need to hold a combinatorial number of states in memory. We will use conservative partitioning heuristics to overcome this common limitation.

By their nature, unreachable states are "don't-care" conditions, and so they can be used to inform logic synthesis about additional optimizations. Our experiments have shown that modern commercial synthesis tools already do some form of reachability analysis in combinational logic, but do not propagate this information across sequential boundaries. We note there have been many prior efforts to enhance synthesis by identifying these types of optimizations. Most recently, the ABC synthesis/verification research tool utilizes a combination of simulation and SAT-sweeping to merge sequentially equivalent nodes in designs, and despite ignoring non-equivalence node relationships, has demonstrated promising area reductions on many benchmark circuits [23]. Our sequential reachability analysis will use conservative approximations to capture more node relationships and thus help eliminate waste in instances

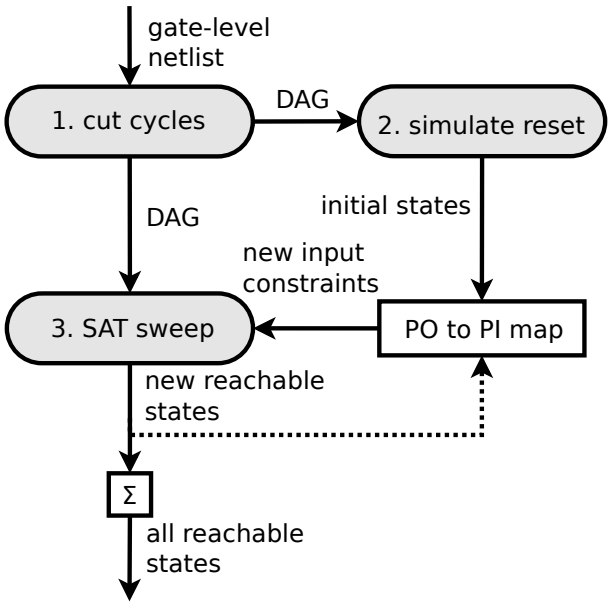


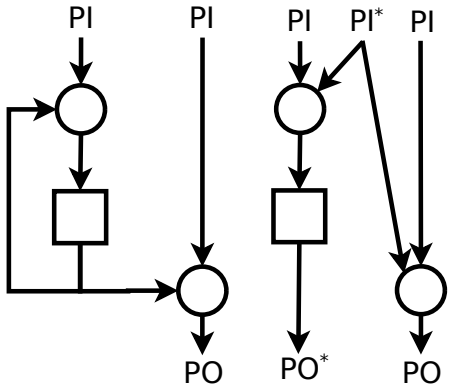
Figure 3.8: High-level depiction of reachability algorithm. The main steps are shaded. The loop (dotted-line) repeats until the set of reachable states reaches a fixed-point solution.

produced from more flexible module generators.

3.2.1 Algorithm

All sequential reachability algorithms tend to take a similar high-level approach: they start from a set of known reachable states and search for any new reachable states, iterating until no new states are reachable. The reset state is a common starting point. While our algorithm does not differ significantly, we include a brief discussion here for completeness. Figure 3.8 graphically depicts our algorithm. The rest of this section describes the main shaded steps in more detail. Sections 3.2.2, 3.2.3, and 3.2.4 present heuristic modifications, unique to our implementation, that allow the algorithm to be practical on real designs with high-level interfaces.

As depicted in Figure 3.7b, our algorithm accepts a gate-level netlist as input, and returns the reachable states for all sequential elements in the design. Generally we found it simpler to parse a gate-level netlist rather than full RTL, so we begin by doing a quick synthesis of our RTL to get a flattened gate-level netlist. The flattened



(a) Initial gate-level netlist (DG) (b) DAG

Figure 3.9: An example of cutting cycles in a sequential netlist (directed graph) to form a directed acyclic graph (DAG). Circular nodes represent combinational standard cells, and square nodes represent sequential standard cells. Note that the new input PI^* and new output PO^* are related.

gate-level netlist is then parsed into a logical directed graph. There are 3 types of nodes in our directed graph: primary inputs (PI), standard cells, and primary outputs (PO). The wires connecting these nodes form the edges of the graph. This graph data structure facilitates logical simulation as well as satisfiability (SAT) analysis.

1. Cut cycles to create a DAG

We first isolate all combinational logic from the sequential elements (flip flops) in the graph, by cutting all edges that are outputs of sequential elements. Figure 3.9 depicts a simple example of this procedure. We connect the output of each sequential element to a new PO node and connect the original fanout of each sequential element to a new PI node. We maintain a lookup table to relate each new PO and PI. The result of these modifications is a directed acyclic graph (DAG), since 1) we have severed all sequential connections and 2) combinational feedback loops are forbidden in standard cell designs.

2. Simulate reset states

Our algorithm requires internal register states (which correspond to some PIs) to be initialized with legal values. We rely on the fact that well-constructed designs have a global reset signal that sets the machine to a known state. We do a logic simulation of all POs, asserting the global reset PI and allowing all other PI nodes to be "don't-care," to automatically determine this initial legal state. Note that this input-to-output simulation is straightforward because we know the logical function of each node (each standard cell), and so it can be accomplished with a single pass through the graph.

3. SAT sweeping

Each main loop iteration starts by seeing if the set of reached PIs has changed. If so, then we do a sweep of SAT calls, one for each of the unreached POs, using the difference in PIs as SAT problem assumptions[2]. Any new satisfiable states are recorded; the loop continues until no new states are found. Note that the number of reachable (unreachable) states will monotonically increase (decrease) as the algorithm runs. For convenience, pseudo-code of this algorithm is included in Appendix B.1.

By default, design inputs are assumed to reach all values. However, external states from the environment can be limited by additionally setting the reachable states of these PI nodes to reflect the desired constraints (not depicted in Figure 3.8).

3.2.2 Heuristic: State Partitioning

The primary concern with the algorithm so far is that the SAT sweeping of unreached states has exponential complexity with the number of POs, so the sweep will have difficulty completing even modestly sized designs (currently, problems arise when a design has more than 20 flop elements). To combat this exponential complexity, we developed conservative heuristics, which we found to work extremely well in practice.

The first such heuristic involves intelligent state partitioning: instead of treating all flops in the design as one large state machine, they can be separated into smaller

[2]See Appendix C for an example of how to formulate a SAT problem.

Table 3.2: Various partitioning examples for a given 4-state, 4-bit FSM.

Scheme	Partitions (by bit)	# partitions	# SAT calls	States reached
A	|3210|	1	16	1,2,13,14
B	|32| |10|	2	8	1,2,13,14
C	|30| |21|	2	8	0-15
D	|3| |2| |1| |0|	4	8	0-15

state machines and treated independently. There are two computational advantages to this approach. First, treating POs independently simplifies their corresponding PI constraints in the SAT problem[3], resulting in faster individual SAT calls. Secondly, and more importantly, the number of required SAT calls is drastically reduced. Imagine the larger problem involves sweeping 10 bits, or $2^{10} = 1024$ total sweeps. Instead, if we break it into 2 different subgroups of 5 bits and sweep each subgroup independently, we only require $2^5 = 32$ sweeps per group, for a total of 64 sweeps instead of 1024. Partitioning in this manner assumes the design reaches the set product of states between the groups while doing a fraction of the work.

Note that any independent grouping of bits in this manner gives a legal conservative result. Since the ultimate goal is to find and remove useless logic associated with states the design cannot reach, it is perfectly okay to think some states are reachable when they are not. In fact, note that synthesis tools inherently assume *all* states are reachable. Hence, this heuristic allows us to trade off sweep time versus efficacy of logic reduction.

Table 3.2 explores partitioning a small 4-bit FSM, using different bit grouping schemes. Scheme A reflects the true partioning while Schemes B, C, and D reflect various smaller paritionings that, for this particular FSM, require fewer SAT calls to sweep. Note that, as expected, all schemes give conservative reachability results (that is, they're all supersets of Scheme A). Also, although Schemes B and C had the same size and number of partitions, B resulted in the optimal reachable set while C did not.

The intuition for this phenomenon is that since this partitioning method will assume the groups are independent, we will get the best quality of results by actually

[3]See Appendix C for details.

picking independent groups. In Table 3.2, the groups in Scheme B are actually independent while in Scheme C they are not. In a generic netlist, it can be difficult to discover these optimal bit groupings without a sophisticated structural analysis. Fortunately, however, this knowledge tends to be embedded within designs already through signal types at the RTL-level (e.g., "reg" or "logic" in Verilog), and can also be identified by instance names at the gate level (assuming the synthesis tool does not obfuscate names). While certainly not perfect, grouping based on signal names intuitively works because they come directly from the designer's intent, and typical "best practice" encourages semantically different signals to be grouped separately for improved clarity and readability.

Once we've identified and partitioned the various sequential groups (SGs), we number the SGs $\{1, 2, 3, ..., g\}$, and determine the set of fan-in PIs for each. We then determine the ideal ordering of SGs that will minimize the total iterations required in the main loop. This step isn't strictly necessary but allows faster convergence of iterative maximum fixed-point solutions [2]. To do this we create a dependency graph among the SGs. The graph has g nodes, one for each group, as well as a root node that represents the original circuit inputs. The directed edges indicate dependencies, i.e., we create an edge AB if an input to SG B is driven by an output of SG A. A reverse postorder traversal of this graph gives us our ideal SG ordering. We note that this is but one approach of partitioning and traversing FSMs, and that there are a number of well-studied variations, of which our described method most closely resembles Cho's MBM method [9].

3.2.3 Heuristic: Sliding Window Algorithm

Despite using the groupings inspired by signal names, we are still likely to end up with some relatively large groups which will be difficult or impossible to sweep (recall more than 20 bits becomes a challenge in our current setup). For example, pipeline registers on data-paths are commonly 32 or 64 bits, and even wide decoded state registers on control-paths can be too large. Although it is easy to either ignore them (assume they reach all states) or arbitrarily divide them into smaller subgroups, both

Table 3.3: Sliding window algorithm results for a sparse example with $n = 8$ and the following 6 reachable states: 0x0F, 0xF0, 0x5A, 0xA5, 0x00, and 0xFF. The algorithm begins with the most significant bit (MSB).

w	s	Max #SAT (per iter)	# Iters	# Candidate states	Total #SAT
8	1	256	1	6	256
6	2	64	2	6	$88 + 6 = 94$
4	4	16	2	16	$32 + 16 = 48$
4	2	16	3	6	$48 + 6 = 54$
2	2	4	4	256	$16 + 256 = 272$
2	1	4	7	256	$28 + 256 = 284$
1	1	2	8	256	$16 + 256 = 272$

of these methods return unsatisfactory results for large control registers with many unreachable states.

The sliding window algorithm attempts to reduce the total work of these large sweeps by first eliminating many states from consideration with little effort, so that the total number of required SAT calls remains low. As an example, let's again consider a 10-bit register that has been divided into two 5-bit groups. If both groups are found to only reach 2 states, then we know the larger group can reach at *most* 4 states (their set product). We can then just do a final SAT pass over those 4 candidate states to find the actual reachable states of the larger group. In this example, we have found the correct answer (with no approximations) using 68 SAT calls instead of 1024. If we generalize this idea beyond mutually exclusive subgroups to a series of overlapping subgroups we get a "sliding window". Each window has size w and step size s, with $w \geq s$. In the example above, $w = s = 5$. A group of size n therefore requires $1 + ceil\left(\frac{n-w}{s}\right)$ iterations to sweep over all bits. At most, each iteration requires 2^w SAT calls; however, when $w > s$ (i.e., there there is overlap between each iteration), the number of required SAT calls per iteration can be reduced whenever there is sparsity in the overlap region. See Appendix B.2 for a pseudo-code implementation of the sliding window algorithm.

Table 3.3 presents the results of running the sliding window algorithm on a sparse 8-bit example. Note that the case of $w = 8, s = 1$ represents the full sweep, requiring 256 SAT calls to find the 6 reachable states. As suggested, there are combinations of

(w, s) that reduce the number of candidate reachable states using far fewer SAT calls. For example, $w = 4, s = 4$ yields 16 candidate states with only 32 SAT calls. The 6 reachable states could then be found by explicitly checking those 16 candidate states, requiring a total of $32 + 16 = 48$ SAT calls. Note that small values of (w, s) fail to reduce the size of the candidate set beyond 256, and so the sliding window algorithm fails to provide any benefit. It is worth pointing out that this example was chosen to be small with $n = 8$ for simplicity and readability. For larger sparse examples (e.g., $n > 20$), the difference in SAT calls between the full sweep and a good sliding window can be many orders of magnitude.

Note that the efficacy of the sliding window heuristic certainly depends on the order in which state bits are grouped (as well as the direction in which the window moves). We observed favorable results in our designs by simply using the bit-orderings defined in the original design, but it is easy to imagine "high-effort" modes that attempt other orderings as well. Furthermore, the optimal values of (w, s) certainly vary on a per-design basis. In practice, we achieved good performance across different designs using $w = 16, s = 8$. All of our examples and implementations start at the most significant bit (MSB).

The sliding window heuristic is only useful when a group's reachable state space is sparse. If the group's state space is more densely populated, then the candidate set obtained from the sliding window algorithm may be fully populated. In this case, we are not willing to actually sweep the full space so we abort, conservatively assuming it reaches all states (allowing us to remove associated SAT assumptions). Intuitively, this is practical for our needs because we are generally interested in understanding control state in flexible designs. Since wide control registers tend to be sparse (e.g., state machines rarely have greater than 2^{20} states), this method lets us solve the groups of interest while ignoring other less-interesting groups.

3.2.4 Heuristic: State-Partitioning for FIFOs

We next turn to removing unused flexibility from generic interfaces. As described in Section 2.1, latency-insensitive interfaces contain both additional control states and

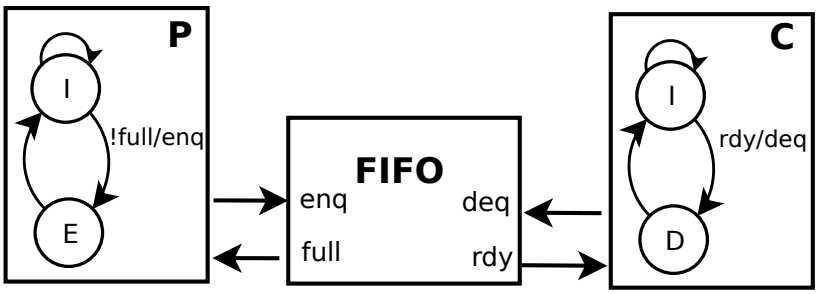


Figure 3.10: Example control state for a simple producer-consumer link implementing a latency-insensitive communication protocol. Note that the consumer can consume at the same rate as the producer can produce, so the FIFO storage isn't necessary.

Table 3.4: Reachable states proved by different state partitionings of the interface in Figure 3.10. Note incorrect enumeration of states E-I, I-D for Scheme A.

Scheme	Partitions	States reached
A	$\|P\|$ $\|C\|$	I-I, **E-I**, **I-D**, E-D
B	$\|PC\|$	I-I, E-D

bypass-FIFO storage elements. Figure 3.10 is a simple example of such a producer-consumer link. Note that the 2-state consumer FSM C always keeps pace with the 2-state producer FSM P, so the FIFO will always be bypassed (making it unnecessary).

However, the state partitioning scheme proposed in Section 3.2.2 fails to identify this relationship because it will analyze the producer and consumer states independently, resulting in extra apparent reachable states. Scheme A in Table 3.4 summarizes the reachability analysis results with this partitioning for this link. Specifically, it finds the "E-I" and "I-D" states are reached, which causes the FIFO to be instantiated instead of bypassed.

Instead, if all producer and consumer states for a given interface are merged into one partition group, our reachability analysis algorithm can prove that the FIFO is never written. This is reflected as Scheme B in Table 3.4, where the "E-I" and "I-D" states are never reached. Since these states are never reached, the FIFO storage states (not shown in Figure 3.10) will never be built.

This simple example suggests a modification to the state partitioning in Section 3.2.2 where the producer-consumer states are merged across a latency-insensitive

interface. This is straightforward if the designer calls out the bypass-FIFO control state (typically a counter), since the producer-consumer states will be fan-in nodes in the state dependency graph.

3.2.5 Logic Optimization

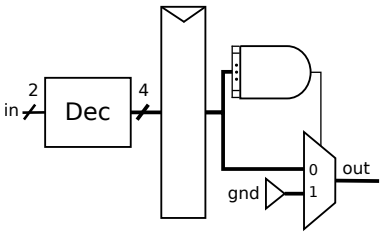
Our algorithm enumerates the reached states of sequential elements, but to use this information we need to either do our own logic optimization and mapping, or annotate those states back into the synthesis tool and leverage its combinational optimization and mapping strengths. Unfortunately, modern tools as yet provide no good way to do this annotation.[4]

To circumvent these issues, we developed a suboptimal solution that demonstrates the value of reachability information for instantiating flexible designs while often reducing the majority of overhead. We manually instantiate pass-through decoders on the outputs of all flop groups in the gate-level netlist, and "program" the pass-through values with the determined reachable states. These pass-through decoders only let certain values appear on the outputs, treating all other conditions as don't-cares. When put through another flattened top-down synthesis flow, the tool will perform the reachability-related logic optimizations within the fanout combinational logic at the expense of the added pass-through decoder.[5] In theory, the pass-through decoders should simply synthesize as wires, and shouldn't add any area. However, some synthesis tools fail to optimally handle the don't-cares, which causes additional logic to be synthesized. If the reductions in logic exceed the added decoder area, the synthesized design will be smaller.

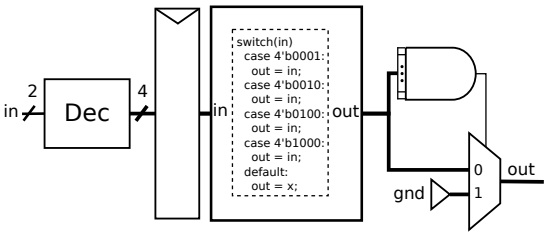
As an example of this method, Figure 3.11a depicts a design with a one-hot decoder, but this one-hot reachability is lost after the flop boundary. By instantiating an additional one-hot pass-through decoder, as in Figure 3.11b, we can force the

[4]Synopsys DesignCompiler's *set_fsm_state_vector* is intended for FSMs with clean feedback logic and often fails on larger designs; moreover it only works with one group at a time. There is planned support for a certain subclass of SystemVerilog assertions, but this is not yet functional and does not allow arbitrary states.

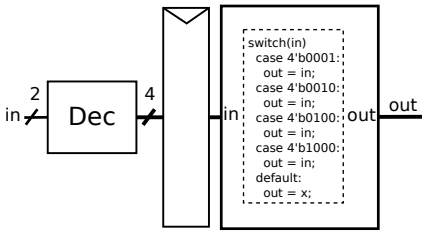
[5]Some synthesis tools ignore decoders wider than 32 bits. We handled these these rare cases by manually injecting the key property that was proven by the reachable states.



(a) A sample design containing a one-hot flopped signal. Note the unneeded multiplexer logic on the output.



(b) The design with an additional pass-through decoder, programmed to pass one-hot signals.



(c) The design with combinational logic optimizations. The pass-through decoder still remains.

Figure 3.11: Using a programmable decoder to annotate a design with reachability information.

synthesis tool to make the desired combinational logic optimizations (Figure 3.11c).

To force synthesis to do the desired logic optimizations we must flatten the design, obscuring the boundaries of this decoder module, which could otherwise be removed. The area result in Figure 3.11c represents an upper bound for this design, because the pass-through decoder contributes overhead.

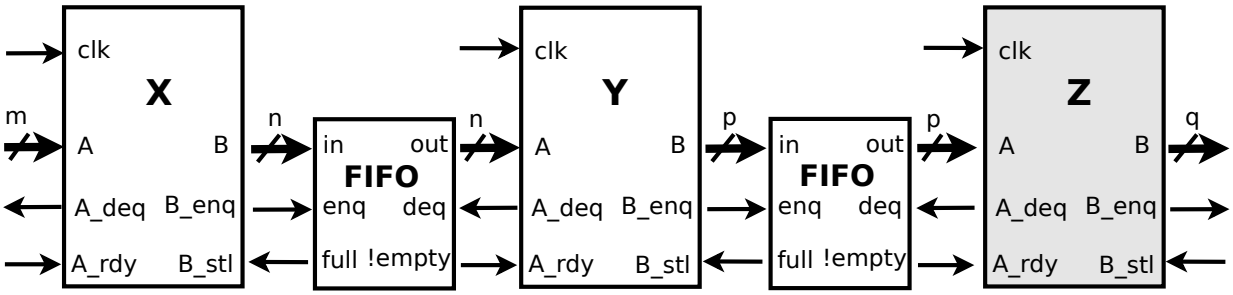
3.2.6 Selective Stage Fusion

In Chapter 2, we showed how to design high-level interfaces by building a superset of state logic to flexibly handle a variety of use-cases. In this chapter, we have presented a reachability analysis technique to remove unused states from design instances, which is always safe because it guarantees cycle-accurate state reachability. In practice, however, this cycle-accurate limitation can be too restrictive. This section discusses common cases where strict reachability analysis can lead to suboptimal results, and proposes a semi-automated workaround.

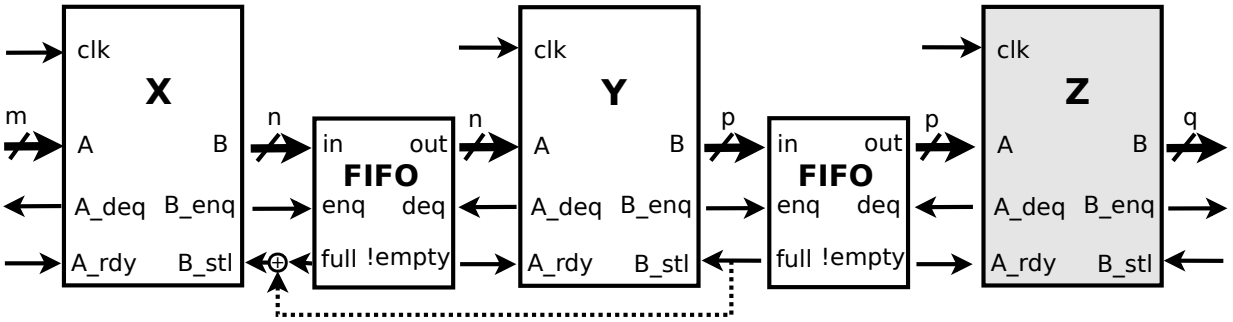
Consider the multi-stage design in Figure 3.12a. Assume that modules X and Y produce and consume at equal rates (similar Figure 3.10's producer/consumer relationship), such that in isolation the interface between them should be optimized. Additionally, assume the shaded module Z consumes at a much slower rate than Y produces, causing the FIFO between Y and Z to eventually fill up. This applies backpressure to Y, causing the FIFO between X and Y to fill up as well. As it should, reachability analysis will observe all of these states, leaving *all* interface overhead intact.

In contrast, a hand-tuned design can be built without flexibility on the XY interface (i.e., the designer can fuse the X and Y modules). Clearly, the fused design is not cycle-accurate with the generic flexible design, but this limitation of reachability analysis does not generally apply for a manual designer. In fact, it is possible that the new fused XY design is superior to the optimized generic design, highlighting a potential shortcoming of our proposed latency-insensitive design approach.

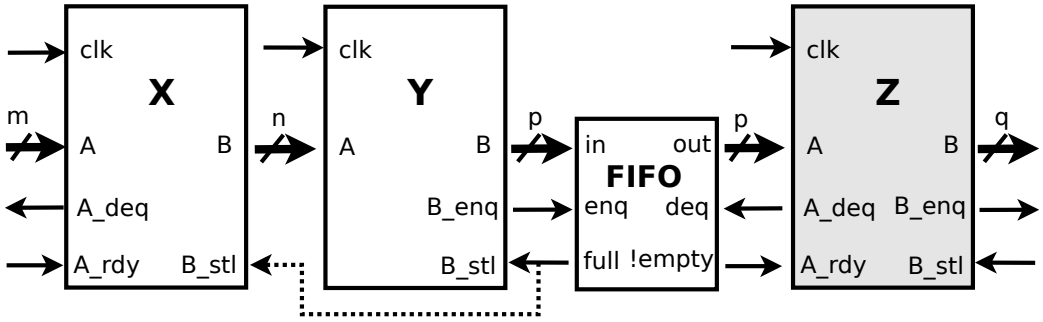
To remedy these cases, we propose a targeted "peephole" optimization to assist a designer in performing this fusion. First, a designer must identify an interface as a



(a) A 3 stage pipeline where modules X and Y operate with identical throughputs, and module Z consumes more slowly. Backpressure causes all FIFOs to eventually fill up, preventing reachability analysis from optimizing any interface.



(b) After a designer identifies the interface XY as a fusion candidate, the design is modified by ORing the downstream backpressure from Y directly into the backpressure at X (dotted line).



(c) Running reachability analysis on the modified design correctly optimizes (fuses) the XY interface, leaving the YZ interface intact.

Figure 3.12: Proposed method to perform selective stage fusion using reachability analysis.

fusion candidate. Second, the design is structurally modified in a systematic manner around the interface (changing the cycle-by-cycle behavior). Third, reachability analysis is run on the modified design. If the final optimized modified design is superior to the optimized generic design, then the modified design should be kept. Otherwise, we conclude that the interface cannot be fused and the change should be reverted.

Figure 3.12b shows the proposed structural modification, assuming the designer has identified the XY interface as a fusion candidate. The modification involves augmenting the backpressure (stall) signal into X with the analogous backpressure signal downstream of Y via a logical OR. This ensures that X will never assert *enq* once the YZ FIFO is full, so the XY FIFO will never be used. Hence, reachability analysis on the modified design is able to optimize (fuse) the XY interface (Figure 3.12c).

This approach has a few obvious limitations. First and foremost, the designer must carefully verify the modified design and ensure potential deadlock conditions (as described in Section 2.1.3 have not been introduced. Moreover, it relies on a designer to choose fusion candidates and the order in which the algorithm is run; poor choices will give poor quality of result. Lastly, it relies on an unmodified downstream backpressure signal, which can prematurely stall and result in increased latency compared to a hand-tuned implementation.

3.2.7 Assumptions and Limitations

The reachability analysis framework presented in this section provides a powerful tool for finding and removing synthesized overheads related to high-level interfaces. However, the framework is subject to a number of known assumptions and/or limitations, which are worth summarizing.

Clock Domain

The algorithms described assume all sequential elements are on the same single global clock domain, and will not work otherwise. Although the following workaround has

not been thoroughly explored, it should be possible to extend to multiple clock domains by running reachability independently on each clock domain, and passing reachability dependencies across domains.

Good Design Practices

As mentioned in Section 3.2.2, our partitioning heuristic relies on the designer to provide meaningful and unique names to sequential elements. For example, bundling semantically different signals onto the same wide bus or register should be avoided. If registers aren't appropriately named, then it is likely that reachability analysis will fail to find any meaningful design optimizations.

Moreover, the algorithm in Section 3.2.1 relies on all sequential elements to have a global reset signal to determine initial states. Note that elements without any explicit reset state (e.g., pipeline registers) are okay, since any initial value can be safely used. Designs with more than one reset domain are generally discouraged by good design practice, and so they haven't been explored with reachability analysis.

Conservative Heuristics

It is worth re-emphasizing that we rely on a number of conservative heuristics to make the reachability algorithm practical and scalable (Sections 3.2.2, 3.2.3, 3.2.4.) Fortunately, these methods are all designed to be conservative by nature, so they will never result in a functionality-incorrect, broken design. However, since they are heuristics, they have merely been observed to work well in practice and we can make no guarantee about optimality or quality of results on new designs.

Chapter 4

High-Level Interfaces in Practice

Chapter 2 explained how to build designs with high-level interfaces, and Chapter 3 showed how existing logic synthesis can be augmented to reduce any resulting implementation overheads. This chapter demonstrates the applicability of these ideas on real-world examples drawn from the Stanford Smart Memories project. We begin with a brief overview of Smart Memories to give better context for our example designs. We then review the individual examples in more detail and explain how each uniquely leveraged high-level interfaces. We then present synthesis results that demonstrate our reachability method can remove most of their overheads, and conclude by analyzing the scalability of our technique.

4.1 Stanford Smart Memories

Stanford Smart Memories is a chip multiprocessor with a memory system flexible enough to support traditional shared memory, streaming, and transactional memory programming models on the same hardware substrate[14][31]. The system was designed to be a multiprocessor whose user could program not only the processors, but the memory system as well.

Figure 4.1 illustrates the Smart Memories hierarchical architecture, which integrates a large number of processors and memory blocks on a single chip. Figure 4.1(c) shows that rather than having explicit instruction and data caches connect to each

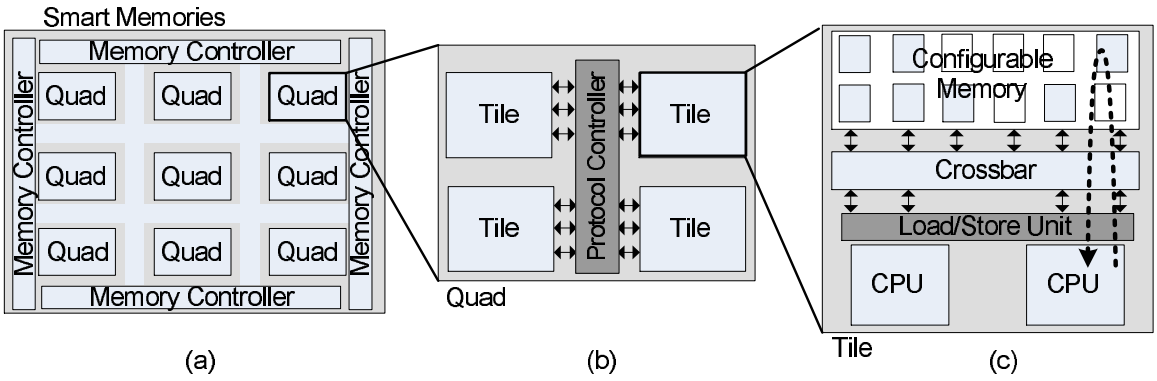


Figure 4.1: Stanford Smart Memories architecture. The mesh (a) is composed of individually fabricated chips called Quads (b). Each Quad contains 4 Tiles (c) and a Memory Protocol Controller (not to be confused with the chip-level Memory Controller, which handles traffic to and from off-chip memory).

processor, the system consists of several memory blocks and a crossbar connecting the memories to the processor cores. In addition to the data arrays, the memories also contain meta-data bits and hardware for implementing particular functionality in the memory systems (e.g., synchronization and cache management). Two VLIW cores and 16 memories are placed in a Tile, and Tiles are placed in groups of four to form Quads (Figure 4.1(b)). The shared Memory Protocol Controller in each Quad provides support for the Tiles by moving data in and out of the local memory blocks and implementing memory protocols (such as cache coherence) in different execution modes. Figure 4.1(a) shows that Quads are then connected to each other and to the off-chip interfaces using Network Routers to form a mesh-like network. External memory controllers are connected to these off-chip interfaces as well.

Our work looks at the effect of flexible interfaces and reachability analysis as applied to each of two major blocks: the Network Router and the Memory Protocol Controller.

4.2 Network Router

The router that was used for chip-to-chip communication in multi-Quad topologies in the Smart Memories project uses both elaboration tables and latency-insensitive

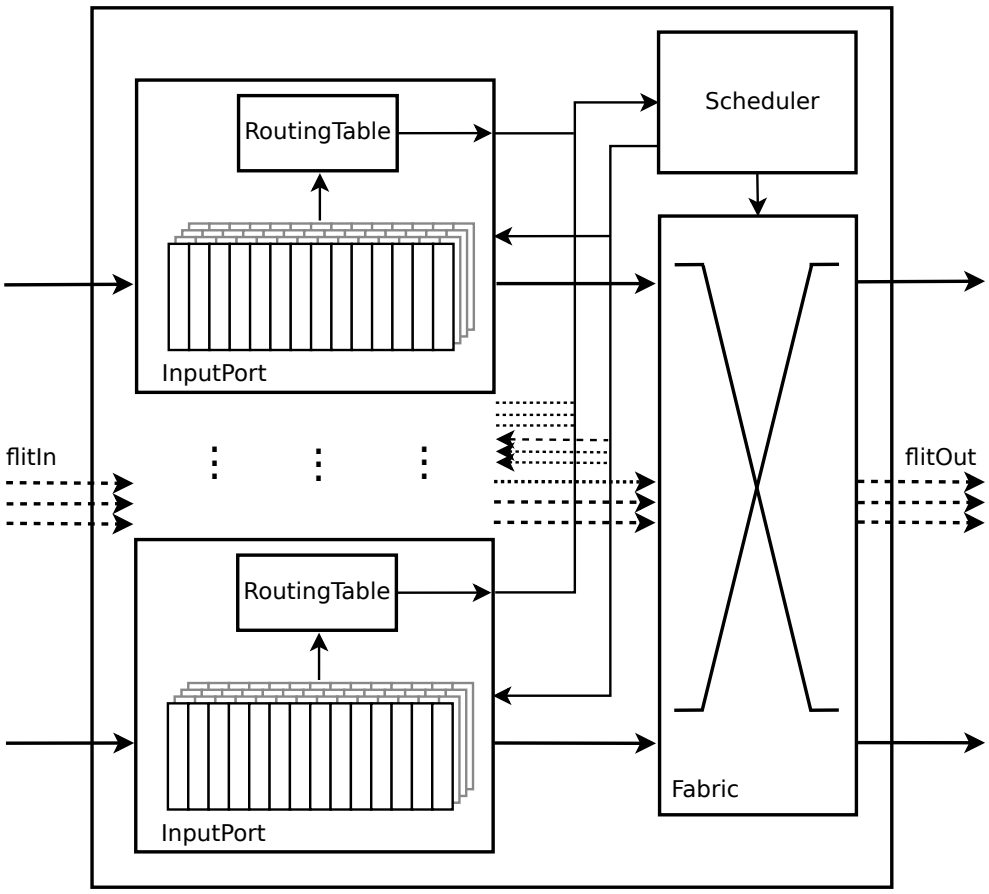


Figure 4.2: A flexible network router design. Note how the RoutingTable interacts with the Scheduler and, indirectly, the Fabric.

interfaces to provide significant design-time flexibility.

The m-by-n input-queued virtual channel router is designed to operate on variable-length packets divided into flits. The minimum-size packet is 1 flit. It forwards flits using cut-through flow control, and supports fanout-splitting multicast. We chose this example because, while not too complex, it demonstrates a number of ways high-level interfaces can be used to increase design flexibility in a practical setting. Figure 4.2 shows a block diagram for the Network Router, and the following subsections describe the microarchitectural units in more detail.

InputPort

The *InputPort* module queues incoming flits per virtual channel, and holds them until ready to send across the *Fabric*. It uses a priority-matrix elaboration parameter to arbitrate among virtual channels, allowing a flexible prioritization scheme.

RoutingTable

The *RoutingTable* unit uses a flexible lookup-table elaboration parameter to determine routing destinations from packet headers. This flexibility allows all combinations of unicast and multicast routing requests from incoming packets. It can also be uniquely programmed per *InputPort*, allowing different routing schemes for different sources.

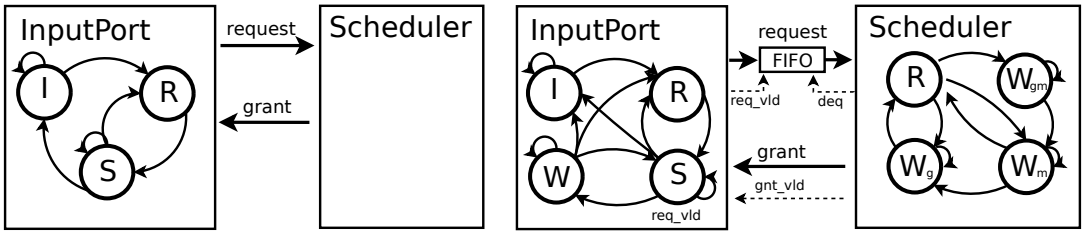
Scheduler

The *Scheduler* unit arbitrates among requests, determining which inputs are granted access to the *Fabric*. Since it is designed to be flexible, it must support all combinations of unicast and/or multicast requests. To prevent system deadlock, it must first arbitrate among overlapping multicast requests so that circular wait dependencies do not occur. If the routing tables are programmed for a unicast-only system, however, this extra arbitration logic is unnecessary and thus becomes an example of logic over-provisioning.

Fabric

The *Fabric* unit is a full crossbar that allows every input to route to every output. Again, depending on the specific routing table configurations, this fully-connected crossbar may be over-provisioned.

To study latency-insensitive interfaces, we created two versions of this router. The first version had fixed timing interfaces, while the second version had a latency-insensitive interface between the *InputPort* and *Scheduler*. Figure 4.3a depicts a



(a) Control logic on one interface within the Network Router. The Scheduler is purely combinational and so does not require any FSM to operate.

(b) The same interface converted to be latency insensitive. Note the added FSM wait (W) states and bypass-FIFO storage.

Figure 4.3: Creating a latency-insensitive interface between the *InputPort* and *Scheduler* modules of the *NetworkRouter*.

simplified interface and the relevant FSM control logic for the interaction of the two modules in a base design where the Scheduler is purely combinational. The *InputPort* state machine transitions between *Idle*, *Route*, and *Schedule* states. The 0-cycle latency of the *Scheduler* block is implicitly assumed in this FSM, and so a *Scheduler* with different latency characteristics will surely break this design.

To decouple these inter-module assumptions, we build control logic to account for latency behavior between blocks. Figure 4.3b depicts the modified interface. The additional *Wait* state in the *InputPort* and the extra valid bits account for a *Scheduler* with greater latency. Note that the *Scheduler* cannot start on a new request until the previous grants have been determined (this ensures the network protocols are adhered to, since continuing flits have priority over new flits). Hence, pipelining the operation of the *Scheduler* is not straightforward in this design, and was not explored. A bypass FIFO queues requests until the FSM in the *Scheduler* is ready.[1] The *InputPort* can never produce more than one outstanding scheduling request, so a FIFO of depth one is sufficient here. In general, however, an explicit backpressure mechanism would be needed to prevent overflow.

The *Scheduler* can start an allocation (dequeue FIFOs) only if all expected continuing requests have been received, and must stall otherwise. This is important

[1]The 3 distinct *Wait* states account for different possibilities of *Scheduler* latency (0 cycles or 1+ cycles) and packet size (single-flit and multi-flit). The packet size distinction allows the network protocol to give continuing flits preference over new flits.

because continuing flits are given priority over new flits in our network. Hence, we use a global stall structure to maintain order among the FSMs from all ports. This structure is notable because it causes the bypass FIFO in each *Scheduler* port to not only depend on its producer and consumer, but on the producers and consumers from *all* other ports as well.

4.2.1 Parameterized Routing

To study the effects of table-based elaboration parameters, we built two versions of the network router. The first version had unicast routing tables, and a hand-tuned unicast *Scheduler*. The second version had unicast routing tables, and a generic *Scheduler* that supported any routing scheme.

4.3 Memory Protocol Controller

The Memory Protocol Controller (PCtrl), previously mentioned in Section 3.1.3 to demonstrate the overhead of runtime configurability, is an example of a complex configurable state machine: shared among four two-processor tiles, it moves data in and out of local memory blocks and implements different memory protocols (such as multiprocessor cache coherence) based on the execution mode. The PCtrl consumes 14% of Quad area, with roughly 200k standard logic cells.

Figure 4.4 shows a high-level view of the PCtrl. It achieves its flexibility through a series of table-based (microprogrammed) controllers. Each of these units has a superset of the functionality required to support a given memory configuration. In most memory configurations, one or more of these tables will be over-provisioned. For example, if all memories are configured in uncached modes, then all microprogram lines and state involving cache operations go unused. Likewise, in cached modes, transactional operations will never be needed.

Unlike the Network Router design in Section 4.2, it is not possible to determine the PCtrl's required functionality based solely on its internal logic. Rather, the types of messages that the PCtrl receives on its ports imply the required functionality. For

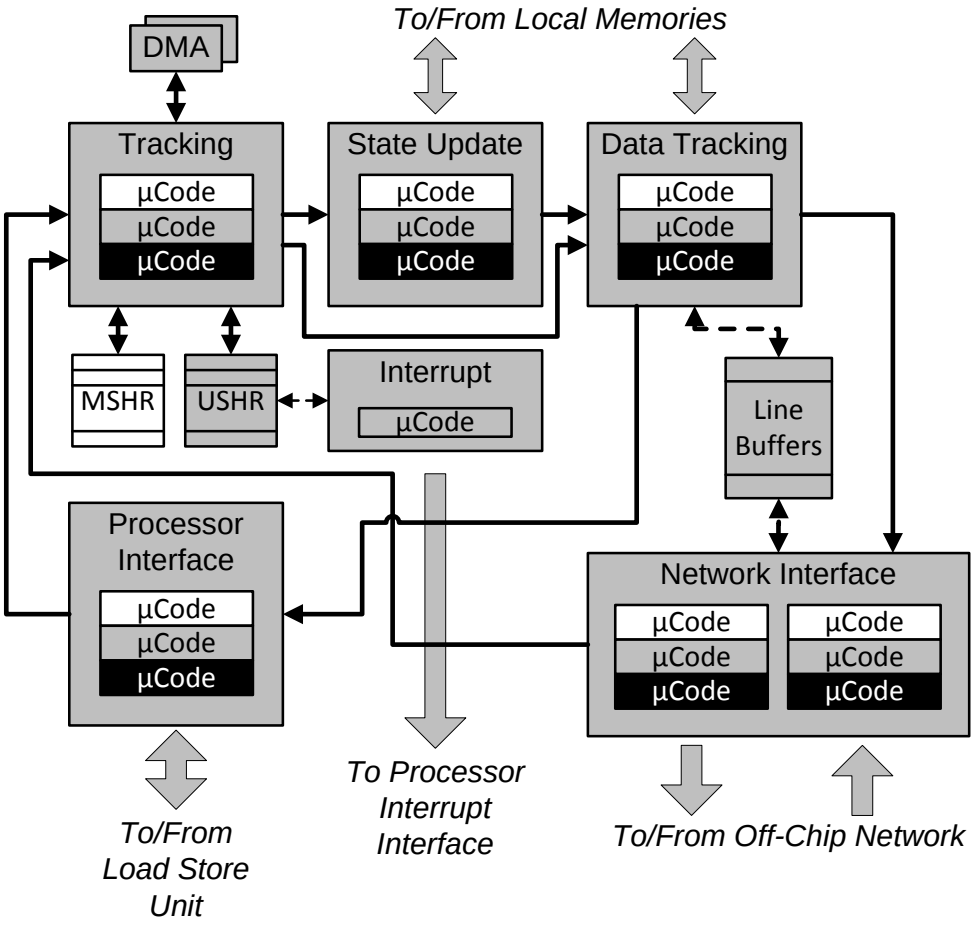


Figure 4.4: Smart Memories Protocol Controller (PCtrl). Blocks shown in white are specific to a cached memory protocol, while those in black are for special memory operations such as transactional memory. Blocks in grey are used by both cached and uncached configurations.

example, an uncached configuration is distinguished by the fact that the processor will never send a "cache miss" request, and the network interface will never send or receive any coherence messages. Therefore, to eliminate unnecessary microcode and structures, the legal states of all inputs have to be considered and propagated through all of the controllers (which are separated by message queues and arbiters). This is similar to analyzing "don't-care" states of inputs imposed by the design environment [10][8].

4.4 Synthesis Results

A graph of all synthesis results for our various designs is presented in Figure 4.5.

We performed synthesis experiments on the flexible Network Router described in Section 4.2 to demonstrate how reachability analysis can remove overhead from flexible design instances. The design was configured with $m = 8$ inputs, $n = 8$ outputs, 6 virtual channels, and 72-bit flits, consuming approximately 30,000 standard logic cells (ignoring large memory queues at the inputs). As mentioned, the Network Router was intentionally forced into an over-provisioned case by using unicast routing tables but keeping all other blocks the same. Figure 4.5 shows the results. For a design targetting a 4.2ns cycle time, these results indicate that the flexible Network Router has a 21% area overhead, but the remaining overhead (after annotation with programmable pass-through decoders) was reduced to 3%.

The IFCx results of Figure 4.5 examine a Network Router with a latency-insensitive interface between the *InputPort* and *Scheduler* modules, as described in Section 4.2, and synthesized to a 4.0ns[2] clock. IFCx represents a router with x total ports (physical and virtual). The routers were configured using a combinational *Scheduler* (from the original custom design) so that the additional interface logic would be unwanted overhead. The areas are normalized to the corresponding custom router with no flexible interface. We are able to remove nearly all of the overhead in these examples because the bypass-FIFO becomes an unused constant and the decoder on the

[2]Note that this Network Router experiment used a more aggressive timing target than the previous one

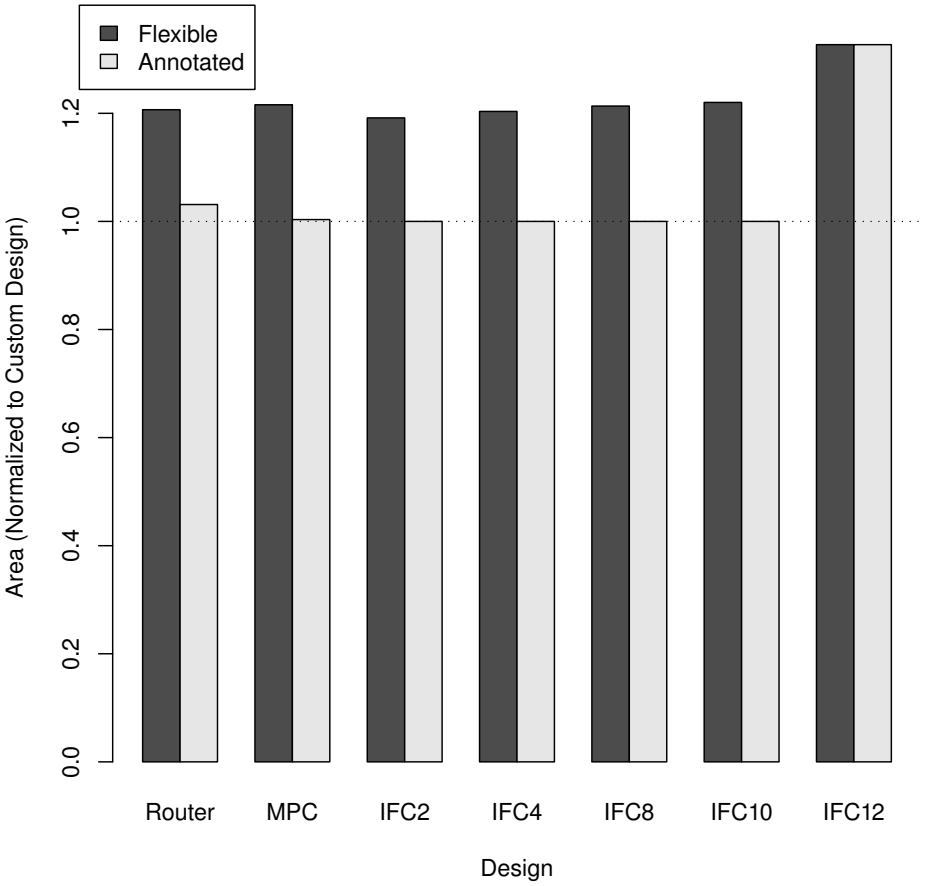


Figure 4.5: Synthesis results for designs with various high-level interface structures synthesized with and without annotating reachability information using programmable pass-through decoders. IFCx represents a Network Router with x total ports. Results are normalized to the corresponding custom design (indicated by the horizontal dashed line).

Table 4.1: Design sizes and algorithm runtimes. *Max* refers to the largest sparse group that exists in each design (number of reachable states / total states).

Design	Gates	Groups	Max	SAT calls	Time
Router	28,512	353	$2^3/2^8$	1.9M	49.7min
IFC2	611	35	$2^4/2^{12}$	73.3k	3.7s
IFC4	2,362	69	$2^8/2^{24}$	279.4k	17.3s
IFC8	3,470	137	$2^{16}/2^{48}$	868.7k	23.6min
IFC10	4,802	171	$2^{20}/2^{60}$	2.6M	21hr
IFC12	6,973	205	$2^{24}/2^{72}$	NA	NA
PCtrl	209,376	5,166	$2^8/2^9$	2.5M	34hr

Scheduler renders the extra control states unnecessary after the tool re-optimizes the logic. Our algorithm failed to optimize IFC12 because the number of merged producer-consumer states exceeded 2^{20}.

After looking at the Network Router, we performed similar experiments on the Protocol Controller (PCtrl), which was configured to only handle uncached memory requests at the inputs, making the original design (which also handles cached and transactional requests) over-provisioned. The PCtrl synthesis results show that the extra area added by the pass-through decoders was insignificant compared to the entire design, so nearly all of the overhead was able to be recovered with our method.

4.5 Scalability of Reachability Algorithm

We implemented our reachability algorithm in Python, using MiniSat-2.2[13] to solve SAT problems. We intended our implementation as a simple proof-of-concept to demonstrate the feasibility of our algorithm, and so it was only optimized until the Python program's runtime was dominated by MiniSat calls. Table 4.1 presents measurements from our code on the various example designs. Runtimes were recorded on a 3GHz Core2 Duo machine with 8GB RAM[3]. Although we used a fixed set of sliding window parameters over all examples, in practice these can be tuned per-design to improve runtimes.

[3]The PCtrl required more RAM than *Router* or *IFCx*; its runtime was measured on a 2.8GHz Opteron with 32GB RAM.

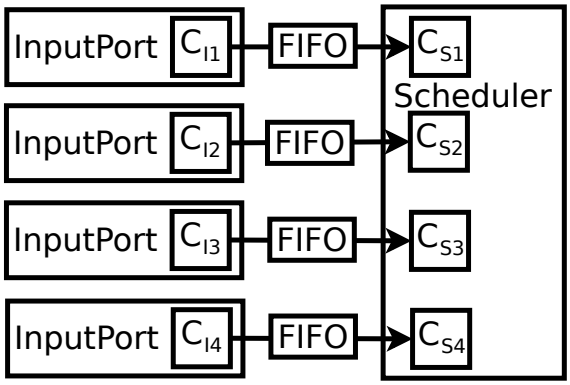


Figure 4.6: A depiction of the interface between *InputPort* and *Scheduler* for a 4-port Router (IFC4). The relevant control logic for each port (see Figure 4.3b for details) is shown.

The runtime numbers demonstrate that our reachability algorithm is feasible for many designs; we found the times were often comparable to that of top-down synthesis.[4] The algorithm takes longer on designs with more groups because these require more SAT calls; furthermore each SAT call takes longer on more complex designs.

The "Max" column refers to the largest sparse group that exists in each design (number of reachable states / total states), which is the most important metric for understanding the limits of our algorithm. Designs with more than 2^{20} reachable states in a sparse group are not feasible to explore, so we cannot remove any overhead associated with that group. However, since it is uncommon to find FSMs with greater than 2^{20} states, we believe this approach is practical on most high-level functionally flexible structures, as evidenced when our algorithm scaled to the 200,000 gate PCtrl.

Latency-insensitive interfaces present an additional challenge because they require artificially merging a port's producer and consumer state machines together into one group. Our method in Section 3.2.4 is a simple way to do this, but can quickly break down with global stall structures like the one described in Section 4.2, particularly since the ports in a router behave independently. This independence causes the reachable space of grouped port states to grow exponentially, quickly becoming infeasibly large. Instead, if additional information could correctly associate states with ports

[4]In fact, the PCtrl example is one of the largest designs we've been able to reliably synthesize in a top-down flow, even without adding our reachability algorithm into the mix.

Table 4.2: Control state groupings for the interface shown in Figure 4.3b. Scheme *A* represents our automated grouping, while Scheme *B* represents user-guided partitioning that separates control state per-port. *Max* refers to the largest sparse group that exists in each case (number of reachable states / total states).

Scheme	Groups	Max
A	$\lvert C_{I1}C_{I2}C_{I3}C_{I4}C_{S1}C_{S2}C_{S3}C_{S4}\rvert$	$2^8/2^{24}$
B	$\lvert C_{I1}C_{S1}\rvert C_{I2}C_{S2}\rvert C_{I3}C_{S3}\rvert C_{I4}C_{S4}\rvert$	$2^2/2^6$

(generally difficult given a flattened gate-level netlist, but often trivial for a designer), we could group states per-port instead of naively grouping all ports, resulting in far fewer required SAT calls (this is similar to the grouping heuristic discussed in Section 3.2.2). Table 4.2 illustrates this concept for IFC4 shown in Figure 4.6. The user-guided partitioning shown in Scheme *B* separates logic on a per-port basis. This partitioning creates many smaller groups, so that the most reachable states of any group remains constant (for any size router). This heuristic gives the correct result in this case because the ports in the Network Router behave independently (by definition, since they are attached to external network sources). Note that this user-guided partitioning solves the otherwise infeasible IFC12 design in under 5 minutes.

Chapter 5

Conclusions

5.1 Overview

We are at a unique period in the evolution of digital system design. The thirst for component flexibility and reuse has never been greater, as these are the best known techniques for managing ever-increasing design complexities. As intuition suggests, many of these techniques sacrifice efficiency (either area, power, or performance) for the benefit of flexibility. Historically, the community has adopted such techniques as long as the design benefits outweigh these costs. One such example was the transition of full-custom design to standard-cell based designs: although standard cell designs are generally inferior to custom designs, they were still adopted because they made designing easier, and designers were typically willing to pay the increases in energy per operation. Unfortunately, since technology scaling has pushed us against a power-wall, energy efficiency is now a primary design constraint. For many modern applications we are less willing to sacrifice energy efficiency to achieve more flexibility in our designs.

High-level interfaces offer a promising solution to this impasse, giving designs greater flexibility without adding implementation overhead. We showed that latency-insensitive communication protocols between modules facilitates system-level design exploration, and reduces the manual tuning required to compose modules into functioning systems. Furthermore, parameterized control-logic offers an efficient approach

to manipulating state machines using simple microprogram modifications. Although these techniques result in overhead using current logic synthesis tools, we demonstrated that automated reachability analysis can remove most (and often all) of this overhead. Hence, high-level interface abstractions can now safely be used in modern designs.

5.2 Future Work

Sections 3.2.2 and 3.2.4 describe simple heuristic methods for partitioning a design into independent machines. Fundamentally, these methods use designer-intent (via register names) to determine independent state groupings, and work well as long as designers adhere to "best-practices". Note that this is similar to the approach taken by current synthesis tools, which require designers to use a specific design style before the tool can identify and perform special FSM synthesis optimizations. However, it would be interesting to explore structural approaches to this problem as well, which wouldn't depend on design styles. Specifically, it might be possible to use feedback relationships in the DAG described in Section 3.2.1 to reliably identify independent state machines.

Another follow-on improvement would be to eliminate the pass-through decoders to inform synthesis about don't-care states, discussed in Section 3.2.5. The decoders were attractive because they work with any modern synthesis tool and don't rely on any special hooks. However, the downside is that tools don't always correctly remove the decoders themselves, occasionally leaving overhead as seen in Section 4.4. A better solution would be explicit directives or hooks in the synthesis tool itself, designed specifically for internal don't-care optimizations, avoiding the need to modify netlist logic.

Although our algorithm for reachability analysis was developed and studied only in the context of optimizing high-level interfaces, it is potentially useful to more general design cases. It would be interesting to explore a wider class of designs and logical structures, since our algorithm should benefit any design that contains unreachable states and wide, partially-encoded signals and buses.

Additionally, modern design "best-practices" encourage designers to liberally utilize RTL assertions (e.g., SystemVerilog includes many complex assertions). Although currently ignored by synthesis, this information could be used to enhance synthesis results, similar to how we use reachability analysis to infer don't-care states.

In the process of this work we found explicit reachability analysis to have interesting (and admittedly unexpected) verification-related side-benefits. Occassionally, the reachable states discovered did not coincide with designer expectations. This generally happened for two different reasons: either the system was actually more complex than the designer anticipated (causing a surprise), or there was a bug somewhere in the design. In both cases the explicit set of reachable states provided valuable insight to designers that would have been otherwise unavailable.

Moreover, as mentioned in Section 3.2.1, our breadth-first reachability analysis algorithm has a lot of similarities with those typically used in formal verification. Although formal verification typically avoids explicitly enumerating states, there still may be possible ways to synergistically combine these two steps to reduce the computational effort required across the overall design flow.

In this work we have shown that it is possible to design RTL components with more generic interfaces without incurring implementation overheads. While our formulation of these high-level interfaces is certainly a useful abstraction for RTL designers, they still require additional effort to implement in each module. An interesting next step would be to embed these concepts into an HLS framework, so that more flexible interfaces could be automatically generated around modules. This is generally difficult using RTL and related meta-languages, since the latency-insensitive protocol often requires modifying internal module control logic. However, since HLS fully captures a module's behavioral rules, it might be possible to modify these rules in a standardized way, automatically generating the new internal control logic. Furthermore, if interface control logic is being automatically generated, it should be possible to also provide automatic partitioning hints to our reachability analysis, improving on the heuristics we developed. For these reasons, we believe high-level interfaces may be even more advantageous in an HLS framework, making future system design even easier.

Appendix A

Verilog Implementations

This appendix presents Verilog implementations of various concepts used throughout this thesis.

A.1 Bypass FIFO

This bypass-capable FIFO can be used to create latency-insensitive interfaces, as described in Section 2.1.

```
module FIFOBypass #(parameter width=8, depth=1)
   (input clk, input reset, input enq, input deq, input [width-1:0] i,
    output logic full, output logic empty, output [width-1:0] o,
    output logic [utils::clog2(depth+1)-1:0] capacity);

   // note: this implements the integer ceiling of the log function (base2)
   parameter clogDepth = utils::clog2(depth);

   logic [width-1:0]                              entry [depth-1:0];

   // read from head address, write to tail address
   logic [clogDepth-1:0]                      head, tail, nextHead, nextTail;


   parameter depthM1 = depth -1;

   wire                           incTail;
   wire                           incHead;
```

```
// tell the world we're empty if we're empty AND no data arriving
assign empty  = (capacity == depth) && (~enq);
assign full   = (capacity == 0    );

// write state iff we have space, and we're not bypassing
assign incTail = enq && (capacity != depth && capacity != 0 ||
                         capacity == 0     &&  deq ||
                         capacity == depth && !deq);

// read state iff we have stuff to read and we're not bypassing
assign incHead = deq && capacity != depth;

// bypass routing logic
assign o = (capacity == depth)?  i : entry[head];

generate
   if (clogDepth == 0) begin
      assign nextHead = '0;
      assign nextTail = '0;
   end
   else begin
      assign nextHead = (head == depthM1[clogDepth-1:0]) ? '0 : head + 1'b1;
      assign nextTail = (tail == depthM1[clogDepth-1:0]) ? '0 : tail + 1'b1;
   end
endgenerate

// store data if queue isn't full OR if deq is active
always @(posedge clk) begin: queue_data
   if (!reset && incTail)
     entry[tail] <= #1 i;
end

// queue capacity register tells how much space is free
always @(posedge clk) begin: cap_logic
   if (reset)
     capacity <= #1 depth;
   else begin
      if (incTail && !deq) begin
         capacity <= #1 capacity - 1;
      end
      else begin
         if (incHead && !enq) begin
            capacity <= #1 capacity + 1;
         end
      end
   end
end
```

```
   // synchronously-resettable D-flip flops (with synchronous write-enable)
   FlopSync #(.width(clogDepth)) headReg
     (.clk(clk), .reset(reset), .en(incHead), .d (nextHead), .q(head));

   FlopSync #(.width(clogDepth)) tailReg
     (.clk(clk), .reset(reset), .en(incTail), .d (nextTail), .q(tail));
endmodule
```

A.2 FSM Styles

This section shows different coding styles for the same 2-input, 1-output, 3-state FSM. Note in A.2.2 and A.2.3 how all unique logic is contained in the elaboration parameters (outside of the module body).

A.2.1 Hardwired

```
module FSM (input clk,
            input  reset,
            input  start,
            input  stop,
            output logic out);
   enum                 {IDLE, PULSE, WAIT}           state, nextstate;

   // state register
   always @(posedge clk)
     if (reset)
       state <= IDLE;
     else
       state <= nextstate;

   // next-state logic
   always_comb begin:ns
      unique case(state)
        IDLE:
          if (start)
            nextstate = PULSE;
          else
            nextstate = IDLE;
        PULSE:
          nextstate = WAIT;
        WAIT:
          if (stop)
            nextstate = IDLE;
          else
            nextstate = WAIT;
        default:
          nextstate = IDLE;
      endcase // unique case (state)
   end // block: ns

   // output logic
   assign out = (state == PULSE);
endmodule // FSM
```

A.2.2 Elaboration Microcode (SystemVerilog)

Note the 2-D elaboration parameters in this implementation are not currently supported by all tools.

```
module FSM #(nStates  = 3,
             nInputs  = 2,
             nOutputs = 1,
             resetState = 0,

             bit [clog2(nStates)-1:0][nInputs+clog2(nStates)-1:0] NEXTSTATE =
             {2'b00, 2'b00, 2'b10, 2'b01, 2'b00, 2'b10, 2'b10, 2'b01,
              2'b00, 2'b00, 2'b10, 2'b00, 2'b00, 2'b10, 2'b10, 2'b00},

             bit [nOutputs-1:0][nInputs+clog2(nStates)-1:0] OUTPUT =
             {1'b0, 1'b0, 1'b1, 1'b0, 1'b0, 1'b0, 1'b1, 1'b0,
              1'b0, 1'b0, 1'b1, 1'b0, 1'b0, 1'b0, 1'b1, 1'b0})

   (input clk,
    input                          reset,
    input [nInputs-1:0]          in,
    output logic [nOutputs-1:0] out);

   logic [clog2(nStates)-1:0] state, nextstate;
   logic [nInputs + clog2(nStates) - 1:0] addr;

   // state register
   always @(posedge clk)
     if (reset)
       state <= resetState;
     else
       state <= nextstate;

   // address is determined by concatenating inputs and state
   assign addr = {in, state};

   // use elaboration parameters as lookup tables
   assign nextstate = NEXTSTATE[addr];
   assign out = OUTPUT[addr];

endmodule // FSM
```

A.2.3 Genesis2 Implementation

```
//; my $NINPUTS  = $self->define_param('NINPUTS' => 2);
//; my $NOUTPUTS = $self->define_param('NOUTPUTS' => 1);
//; my $NSTATES  = $self->define_param('NSTATES' => 3);
//; my $NS_MEM = $self->define_param('NS_MEM'=>[0,2,2,0,0,2,0,0,1,2,2,0,1,2,0,0]);
//; my $OUT_MEM = $self->define_param('OUT_MEM'=>[0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0]);
//; my $RESET_STATE = $self->define_param('RESET_STATE' => 0);

module FSM
  (input clk,
   input                           reset,
   input [`$NINPUTS-1`:0]           in,
   output logic [`$NOUTPUTS-1`:0] out);

   logic [`clog2($NSTATES)-1`:0]  state, nextstate;
   logic [`$NINPUTS + clog2($NSTATES)-1`:0] addr;

   logic [`clog2($NSTATES)-1`:0][`$NINPUTS+clog2($NSTATES)-1`:0] ns_mem;
   logic [`$NOUTPUTS-1`:0][`$NINPUTS+clog2($NSTATES)-1`:0]          out_mem;

   // initialize memories with constants
   //; my $idx = 0;
   //; foreach my $val (@NS_MEM){
   assign ns_mem[`$idx`] = `$val`;
   //; $idx++; }
   // $idx = 0;
   //; foreach my $val (@OUT_MEM) {
   assign out_mem[`$idx`] = `$val`;
   //; $idx++; }

   // state register
   always @(posedge clk)
     if (reset)
       state <= `$RESET_STATE`;
     else
       state <= nextstate;

   // address is determined by concatenating inputs and state
   assign addr = {in, state};

   // use constant tables to do lookup
   assign nextstate = ns_mem[addr];
   assign out = out_mem[addr];

endmodule // FSM
```

Appendix B

Pseudo-code

B.1 Reachability analysis: main loop

```
updated = true;
while (updated) {
  updated = false;
  foreach grp in ordered-group-list {
    // inputs are outputs of other groups
    currIn = dag.getInputs(grp, reached);

    // check for any updates
    if (prevIn[grp] != currIn) {
      // we only care about new input states
      newIn = diff(currIn, prevIn[grp]);
      prevIn[grp].add(currIn);

      // unreached states are complement
      unreached = ~reached[grp];

      // do a SAT analysis on the unreached states
      // using the sliding window algorithm to avoid
      // sweeping large groups
      newReached = dag.swSAT(grp, unreached, newIn);

      // only update if new states were reached
      if (newReached.size() > 0) {
        updated = true;
        reached[grp].add(newReached);
      }
    }
```

```
  }
}
```

B.2 Reachability analysis: sliding window

```
// find the reachable states of grp in dag
// using the sliding-window algorithm
// unreached are the states to be explored
// inputs are the reachable states of inputs
function swSAT(dag, grp, unreached, inputs) {

  // set algorithm parameters
  group = dag.getGroup(grp);
  n = group.size();
  w = min(n,16);
  s = 8;
  MAX_STATES = 2**19;

  // initialize variables
  candidates = group.states;
  nWindows=1+(n-w)/(s);
  num_states = 0;

  for (i=0; i < nWindows; i++) {
    // list of bits in this window
    bits_i = group.bits(i*s, i*s+w-1);
    // list of bits not in this window
    bits_i_c = group.bits() - bits_i;

    // unreached states for this window are the
    // complement of the subset of possibly reached states
    unreached_i = ~candidates.subset(bits_i);

    // run SAT sweep on this window over unreached states
    reached_i = group.sweepSAT(bits_i, unreached_i, inputs);

    // retrieve reached states outside of this window
    reached_i_c = candidates.subset(bits_i_c);

    // count the number of new candidate states
    // (it will be the set-product)
    num_states += reached_i_c.size() * reached_i.size();

    // abort if we ever exceed our state limit
    if (num_states > MAX_STATES) {
      return null;
```

```
    }

    // keep the newly reached states
    candidates.add(reached_i);
  }

  // now do a final pass over all remaining candidate states
  return group.sweepSAT(group.bits(), candidates, inputs);
}
```

Appendix C

Using SAT Solver

C.1 SAT Solver Input

Our SAT solver accepts input in conjunctive normal form (CNF). CNF consists of a conjunction (logical AND) of many "clauses", where each "clause" is a disjunction (logical OR) of variables. Individual variables can be inverted within each clause. This section describes an efficient process for converting a logical expression into a product-of-sums (CNF).

File Format

Each line in a CNF file is a clause, containing variable names separated by spaces. Additionally, each clause ends with a special "0" token. Inversion is indicated with a "-" character preceding a variable name.

Conversion Procedure

Our procedure consists of traversing our DAG (see Section 3.2.1) in a reverse depth-first manner, starting at the flip flops of interest (this is similar to walking the parse tree of a logical boolean expression). Each newly visited node is converted to CNF clauses. When finished, the combination of all CNF clauses from all nodes describes the logical behavior of our circuit.

To convert each node (logic gate) to CNF, we first parse the node's Boolean logic to create an abstract syntax tree (AST) consisting of unary and binary operators. The unary operators we support are *assignment* and *inverse*, and the binary operators are *AND* and *OR*. Note that these are sufficient to support any Boolean logic expression. One simple way of parsing an expression into an AST is to express the logic of each node as a Boolean expression in Python, and then use the built-in *ast* module to create the tree. The following subsections show the CNF clauses that are generated for these simple operators.

Note that all node outputs (as well as circuit inputs) are given unique variable names in the final CNF. Additionally, we generate temporary variables as needed for complex gates.

assignment: $y := a$

```
-y a 0
-a y 0
```

inverse: $y :=!a$

```
-y -a 0
 a  y 0
```

AND: $y := a * b$

```
 a -y 0
 b -y 0
-a -b y 0
```

OR: $y := a + b$

```
-a y 0
-b y 0
 a b -y 0
```

Input/Output Constraints

Additionally, our SAT solver accepts optional variable constraints (or assumptions) in disjunctive normal form (DNF). DNF consists of a disjunction (logical OR) of many "clauses", where each "clause" is a conjunction (logical AND) of variables. Individual variables can be inverted within each clause.

It is straightforward to use variable assumptions for constraining inputs to known reachable states, since each reachable state is simply a new DNF clause (with each variables represting an input).

Before running SAT on the CNF in Section C.1, we must constrain the output variables to reflect the state being queried. Fortunately, these output constraints are easy to add to the CNF, since each bit in a particular state will become a new clause.

For example, if variables $\{1, 2, 3, 4\}$ are variables that represent register outputs, and we wish to query whether state 0xd is reachable, then we would add the following 4 output clauses to the CNF:

```
1 0
2 0
-3 0
4 0
```

C.2 Example

Figure C.1 shows an example DAG circuit that will illustrate our SAT problem formulation..

A CNF representation of Figure C.1. Note the addition of an extra variable *t0* to handle the complex gate *g2*.

```
g2 -y1 0
rb -y1 0
-g2 -rb y1 0
-g2 -t0 0
g2 t0 0
```

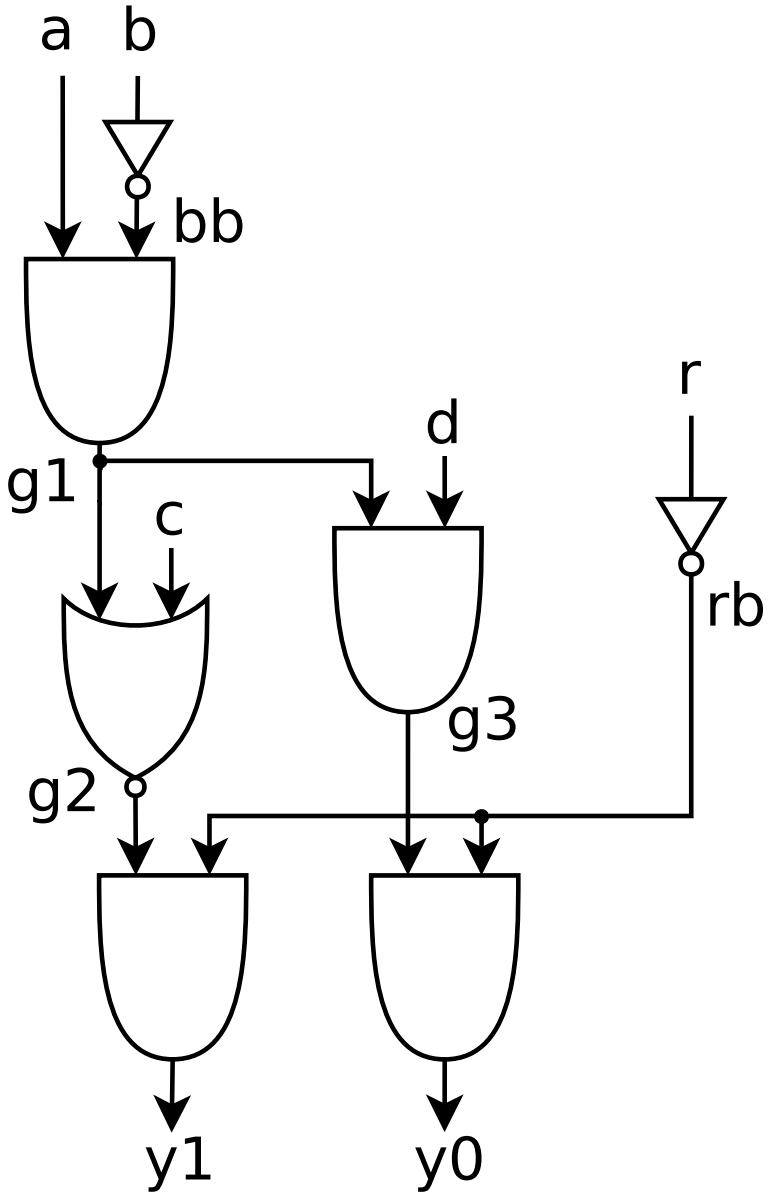


Figure C.1: An example DAG circuit with 4 inputs (a, b, c, d, r) and 2 outputs ($y0$ and $y1$). All gate output labels are shown as well.

```
-g1 t0 0
-c t0 0
g1 c -t0 0
a -g1 0
bb -g1 0
-a -bb g1 0
-bb -b 0
bb b 0
-rb r 0
rb r 0
g3 -y0 0
rb -y0 0
-g3 -rb y0 0
g1 -g3 0
d -g3 0
-g1 -d g3 0
```

To sweep the 4 potential output states, we would append the following new clauses to the CNF in the previous section before running the SAT solver. SAT results are shown.

{y1,y0} == 2'b00

```
-y0 0
-y1 0
```

result: SATISFIABLE

{y1,y0} == 2'b01

```
y0 0
-y1 0
```

result: SATISFIABLE

{y1,y0} == 2'b10

```
-y0 0
y1 0
```

result: SATISFIABLE

{y1,y0} == 2'b11

```
y0 0
y1 0
```

result: UNSATISFIABLE

Bibliography

[1] Open SystemC Initiative (OSCI). http://www.systemc.org/home. IEEE Std. 1666-2005.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley, 2nd edition, 2007.

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In Proceedings of the 49th Annual Design Automation Conference, DAC '12, pages 1216–1225, New York, NY, USA, 2012. ACM.

[4] S. M. Bauer. Bell Labs microcode for the IBM 360/67. In Proceedings of the 8th annual workshop on Microprogramming, MICRO 8, pages 40–44, New York, NY, USA, 1975. ACM.

[5] Luca P. Carloni, Kenneth L. McMillan, Alexander Saldanha, and Alberto L. Sangiovanni-Vincentelli. A methodology for correct-by-construction latency insensitive design. In Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design, ICCAD '99, pages 309–315, Piscataway, NJ, USA, 1999. IEEE Press.

[6] Luca P. Carloni, Kenneth L. Mcmillan, and Alberto L. Sangiovanni-vincentelli. Latency Insensitive Protocols. In Computer Aided Verification, pages 123–133. Springer Verlag, 1999.

[7] Luca P. Carloni and Alberto L. Sangiovanni-Vincentelli. Performance analysis and optimization of latency insensitive systems. In Proceedings of the 37th Annual Design Automation Conference, DAC '00, pages 361–367, New York, NY, USA, 2000. ACM.

[8] Kai-hui Chang, V. Bertacco, and I.L. Markov. Customizing IP cores for system-on-chip designs using extensive external don't-cares. In Design, Automation Test in Europe (DATE) Conference Exhibition, 2009., pages 582 –585, 2009.

[9] Hyunwoo Cho, Gary D. Hachtel, Enrico Macii, Bernard Plessier, and Fabio Somenzi. Algorithms for approximate fsm traversal. In Proceedings of the 30th international Design Automation Conference, DAC '93, pages 25–30, New York, NY, USA, 1993. ACM.

[10] Hong-Zu Chou, Kai-Hui Chang, and Sy-Yen Kuo. Optimizing blocks in an soc using symbolic code-statement reachability analysis. In Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific, pages 787 –792, 2010.

[11] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In Proceedings of the international workshop on Automatic verification methods for finite state systems, pages 365–373, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[12] David L. Dill. The Murphi Verification System. In Computer Aided Verification. 8th International Conference, pages 390–393. Springer-Verlag, 1996.

[13] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, SAT, volume 2919 of Lecture Notes in Computer Science, pages 502–518. Springer, 2003.

[14] Amin Firoozshahian, Alex Solomatnikov, Ofer Shacham, Zain Asgar, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. A Memory System Design Framework: Creating Smart Memories. In

Proceedings of the 36th Annual International Symposium on Computer Architecture, ISCA '09, pages 406–417, New York, NY, USA, 2009. ACM.

[15] David Gifford and Alfred Spector. Case study: IBM's system/360-370 architecture. Communications of the ACM, 30(4):291–307, April 1987.

[16] Mark Horowitz. Why Design Must Change: Rethinking Digital Design. http://www.synopsys.com/apps/community/university/video/rethinking_digital_design.html, 2010.

[17] IEEE. IEEE Standard Verilog Hardware Description Language, 1364-2001 edition, September 2001.

[18] IEEE. IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language, 1800-2009 edition, 2009.

[19] Kyle Kelley, Megan Wachs, Andrew Danowitz, P. Stevenson, S. Richardson, and Mark Horowitz. Intermediate representations for controllers in chip generators. In DATE, pages 1394–1399. IEEE, 2011.

[20] Kyle Kelley, Megan Wachs, John Stevenson, Stephen Richardson, and Mark Horowitz. Removing overhead from high-level interfaces. In Proceedings of the 49th Annual Design Automation Conference, DAC '12, pages 783–789, New York, NY, USA, 2012. ACM.

[21] Jason Leonard and William H. Mangione-Smith. A case study of partially evaluated hardware circuits: Key-specific des. In FPL '97: Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications, pages 151–160, London, UK, 1997. Springer-Verlag.

[22] N. McKay, T. Melham, K. W. Susanto, and S. Singh. Dynamic Specialization of XC6200 FPGAs by Partial Evaluation. In FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, page 308, Washington, DC, USA, 1998. IEEE Computer Society.

[23] Alan Mishchenko, Michael Case, Robert Brayton, and Stephen Jang. Scalable and scalably-verifiable sequential synthesis. In Proc. 2008 IEEE/ACM Int'l Conf on Computer-Aided Design, ICCAD '08, pages 234–241, Piscataway, NJ, USA, 2008. IEEE Press.

[24] Gordon Moore. Cramming More Components onto Integrated Circuits. Electronics Magazine, 38(8), April 1965.

[25] Madhubanti Mukherjee and Ranga Vemuri. A novel synthesis strategy driven by partial evaluation based circuit reduction for application specific dsp circuits. In ICCD '03: Proceedings of the 21st International Conference on Computer Design, page 436, Washington, DC, USA, 2003. IEEE Computer Society.

[26] R. Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on, pages 69 – 70, june 2004.

[27] NVIDIA. FERMI Compute Architecture Whitepaper. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009.

[28] A. Padegs. System/360 and beyond. IBM J. Res. Dev., 25(5):377–390, September 1981.

[29] David A. Patterson and John L. Hennessy. Computer organization and design (2nd ed.): the hardware/software interface. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[30] Ofer Shacham. Chip Multiprocessor Generator: Automatic Generation Of Custom and Heterogeneous Compute Platforms. PhD thesis, Stanford University, 2010.

[31] Ofer Shacham, Zain Asgar, Han Chen, Amin Firoozshahian, Rehan Hameed, Christos Kozyrakis, Wajahat Qadeer, Stephen Richardson, Alex Solomatnikov,

Don Stark, Megan Wachs, and Mark Horowitz. Smart memories polymorphic chip multiprocessor. In Proceedings of the Design Automation Conference, 2009.

[32] Ofer Shacham, Omid Azizi, Megan Wachs, Wajahat Qadeer, Zain Asgar, Kyle Kelley, John P. Stevenson, Stephen Richardson, Mark Horowitz, Benjamin Lee, Alex Solomatnikov, and Amin Firoozshahian. Rethinking Digital Design: Why Design Must Change. IEEE Micro, 30:9–24, 2010.

[33] Ofer Shacham, Sameh Galal, Sabarish Sankaranarayanan, Megan Wachs, John Brunhaver, Artem Vassiliev, Mark Horowitz, Andrew Danowitz, Wajahat Qadeer, and Stephen Richardson. Avoiding game over: bringing design to the next level. In Proceedings of the 49th Annual Design Automation Conference, DAC '12, pages 623–629, New York, NY, USA, 2012. ACM.

[34] Daniel J. Sorin, Manoj Plakal, Anne E. Condon, Mark D. Hill, Milo M. K. Martin, and David A. Wood. Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. IEEE Trans. Parallel Distrib. Syst., 13(6):556–578, 2002.

[35] Blaine Stackhouse, Brian Cherkauer, Mike Gowan, Paul Gronowski, and Chris Lyles. A 65nm 2-Billion-Transistor Quad-Core Itanium Processor. In International Solid State Circuits Conference. IEEE, 2008.

[36] Alexander Stepanov and Meng Lee. The standard template library. Technical report, WG21/N0482, ISO Programming Language C++ Project, 1994.

[37] D. Stoffel, M. Wedler, P. Warkentin, and W. Kunz. Structural fsm traversal. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 23(5):598 – 619, May 2004.

[38] Synopsys. Design Compiler Optimization Reference Manual, March 2010.

[39] Herv J. Touati, Hamid Savoj, Bill Lin, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using bdds. In ICCAD'90, pages 130–133, 1990.

[40] S. G. Tucker. Microprogram control for system/360. IBM Syst. J., 6(4):222–241, December 1967.

[41] Muralidaran Vijayaraghavan. Private Communication, 2011.

[42] Muralidaran Vijayaraghavan and Arvind Arvind. Bounded dataflow networks and latency-insensitive circuits. In Proceedings of the 7th IEEE/ACM international conference on Formal Methods and Models for Codesign, MEMOCODE'09, pages 171–180, Piscataway, NJ, USA, 2009. IEEE Press.

[43] M.V Wilkes and J.B. Stringer. Micro-programming and the design of the control circuits in an electronic digital computer. In Mathematical Proceedings of the Cambridge Philosophical Society.

[44] Xilinx. Virtex-II Pro and Virtex-II Pro X FPGA User Guide, November 2007.